

SAS/C[®]
Development
System
Library Reference



**SAS/C[®] Development System
Library Reference**
Version 6.0



■ **SAS/C[®] Development System Library Reference**

Version 6.0



SAS Institute Inc.
SAS Campus Drive
Cary, NC 27513

The correct bibliographic citation for this manual is as follows: SAS Institute Inc., *SAS/C* Development System Library Reference, Version 6.0*, Cary, NC: SAS Institute Inc., 1992. 589 pp.

SAS/C* Development System Library Reference, Version 6.0

Copyright © 1992 by SAS Institute Inc., Cary, NC, USA.

ISBN 1-55544-495-4

All rights reserved. Printed in the United States of America. No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form or by any means, electronic, mechanical, photocopying, or otherwise, without the prior written permission of the publisher, SAS Institute Inc.

1st printing, June 1992

The SAS® System is an integrated system of software providing complete control over data access, management, analysis, and presentation. Base SAS software is the foundation of the SAS System. Products within the SAS System include SAS/ACCESS®, SAS/AF®, SAS/ASSIST®, SAS/CPE®, SAS/DMI®, SAS/ETS®, SAS/FSP®, SAS/GRAPH®, SAS/IML®, SAS/IMS-DL/I®, SAS/OR®, SAS/QC®, SAS/REPLAY-CICS®, SAS/SHARE®, SAS/STAT®, SAS/CALC®, SAS/CONNECT®, SAS/DB2®, SAS/EIS®, SAS/ENGLISH®, SAS/INSIGHT®, SAS/LAB®, SAS/LOOKUP®, SAS/NVISION®, SAS/PH-Clinical®, SAS/SQL-DS®, SAS/TOOLKIT®, and SAS/TUTOR® software. Other SAS Institute products are SYSTEM 2000® Data Management Software, with basic SYSTEM 2000, CREATE®, Multi-User®, QueX®, Screen Writer®, and CICS interface software; NeoVisuals® software; JMP®, JMP IN®, JMP SERVE®, and JMP Ahead® software; SAS/RTERM® software; and the SAS/C® Compiler and the SAS/CX® Compiler. MultiVendor Architecture™ and MVA™ are trademarks of SAS Institute Inc. SAS Institute also offers SAS Consulting® and On-Site Ambassador™ services. SAS Communications®, SAS Training®, SAS Views®, the SASware Ballot®, and Observations™ are published by SAS Institute Inc. All trademarks above are registered trademarks or trademarks of SAS Institute Inc. in the USA and other countries. ® indicates USA registration.

The Institute is a private company devoted to the support and further development of its software and related services.

Amiga Includes and Development Tools Version 1.3

Copyright © 1985, 1986, 1987, 1988 Commodore-Amiga, Inc. All rights reserved.

Amiga Includes and Development Tools Version 2.0

Copyright © 1990 Commodore-Amiga, Inc. All rights reserved.

Amiga Workbench™ Version 1.3

Copyright © 1985, 1986, 1987, 1988 Commodore-Amiga, Inc. All rights reserved.

Amiga Workbench™ Version 2.0

Copyright © 1988, 1990 Commodore-Amiga, Inc. All rights reserved.

Amiga®, AmigaDOS®, Intuition®, Kickstart®, and Workbench™ are registered trademarks or trademarks of Commodore-Amiga, Inc. Commodore® is a registered trademark or trademark of Commodore Electronics Limited.

Other brand and product names are registered trademarks or trademarks of their respective companies.

Contents

v Reference Aids

vii Credits

ix Acknowledgments

xi Using This Book

Page	1	Chapter 1 What Libraries Can I Access?
	1	Introduction
	1	Using the Shared Libraries
	2	Using the Linkable Libraries
	3	Creating Your Own Libraries
	5	Chapter 2 Using the Commodore Libraries
	5	Introduction
	5	Using amiga.lib
	6	Using the Prototype Files
	7	Defining Library Bases
	11	Chapter 3 Using the SAS/C Libraries
	11	Introduction
	11	Determining Which Library to Use
	12	Compiling with the Correct Library
	13	Linking with the Correct Libraries
	15	Using the Math Libraries
	19	Chapter 4 Using the System Header Files to Access Libraries
	19	Introduction
	19	Compressing Header Files
	20	Using Built-in Functions
	21	Using Global Symbol Tables
	25	Chapter 5 Advanced Library Techniques
	25	Introduction
	25	Using #pragma Statements
	31	Creating and Using Your Own Libraries

41 Chapter 6 **Predefined Data Name Reference**

83 Chapter 7 **Library Reference**

84 The `__STRICT_ANSI__` Symbol

84 Function Categories

565 **Index**

Reference Aids

Tables

8	2.1	Library Bases for .library Files
9	2.2	Library Bases for .device Files
10	2.3	Library Bases for .resource Files
14	3.1	Options Required to Compile and Link with Each Library
27	5.1	Register Conventions

Credits

Documentation

Composition	Candy R. Farrell, Denise T. Jones, Blanche W. Phillips, Pamela A. Troutman
Graphic Design	Creative Services Department
Proofreading	Patsy P. Blessis, Heather B. Dees, Laurie J. Medley, Karen A. Olander, Josephine P. Pope, Christian Schwoerke, John M. West
Writing and Editing	Jeanne Ferneyhough, N. Elizabeth Malcom, Patricia Glasgow Moell

Software

Development	Steve Krueger, Douglas Walker, Michael S. Whitcher
Technical Support	Jim Cooper, Khristi Tomlinson
Testing	Alan R. Beale, Twilah K. Blevins

Acknowledgments

Many people make significant and continuing contributions to the development of the SAS/C Development System. First among them is John A. Toebe VIII, who was instrumental in the development of Versions 4.0 and 5.0 and who directed the early development of Version 6.0.

Others who have contributed to the development and testing of Version 6.0 of the SAS/C Development System include Ralph Babel, Bruce M. Drake, Nathan S. Fish, Maximilian Hantsch, Randell Jesup, David N. Junod, Andrew Kopp, Martin J. Laubach, Christian Ludwig, Andrew N. Mercier, Mike Meyer, Larry Rosenman, Carolyn Scheppner, Steve Shireman, Michael Sinz, Martin Taillefer, Steve Tibbett, John Wiederhahn, Loren Wilton, and Kenneth Yarnall.

The final responsibility for the SAS/C Development System lies with SAS Institute alone. We hope that you will always let us know your feelings about our product and its documentation. It is through such communication that the progress of SAS software continues to be accomplished.

Using This Book

Purpose

SAS/C Development System Library Reference, Version 6.0 describes the content and use of each of the libraries available to the SAS/C programmer under AmigaDOS. Some of these libraries are provided by Commodore, and some are provided by the SAS/C Development System.

“Using This Book” describes how you can best use this book. It describes the book’s intended audience, the audience’s prerequisite knowledge, the book’s organization and its conventions, and additional documentation that is available to you.

Audience

The information in this book is written for experienced C programmers. This book assumes you have a working knowledge of C and AmigaDOS. For a list of publications you can use to learn C or AmigaDOS, refer to the section “Additional Documentation” at the end of “Using This Book”.

How to Use This Book

This section gives an overview of the book’s organization and content. The book’s chapters are described, followed by a section on how to use each chapter.

Organization *SAS/C Development System Library Reference* is organized into seven chapters:

Chapter 1, “What Libraries Can I Access”

describes the two major types of libraries and lists all of the libraries you can access from your SAS/C programs.

Chapter 2, “Using the Commodore Libraries”

describes how to access the shared libraries provided by Commodore.

Chapter 3, “Using the SAS/C Libraries”

describes each of the libraries provided by the SAS/C Development System. It also explains which compiler options enable you to access these libraries.

Chapter 4, “Using the System Header Files to Access Libraries”

describes how to use header files and built-in functions to speed up the compilation and reduce the size of your program.

Chapter 5, “Advanced Library Techniques”

describes how to create your own shared or linkable libraries and how to create and use your own **#pragma** statements.

Chapter 6, “Predefined Data Name Reference”

describes each of the data names provided by the SAS/C Development System.

Chapter 7, “Library Reference”

describes each of the library functions provided by the SAS/C Development System.

What You Should Read

The following table points you to specific chapters in this book for information on particular topics. For other references, refer to “Additional Documentation” later in this section.

For information on	You should read
major types of available libraries	Chapter 1
accessing libraries	Chapters 2 and 3
reducing the amount of time required to compile your program or reducing the amount of disk space required to store your program	Chapter 4
creating your own libraries	Chapter 5
specific library functions or data names	Chapters 6 and 7

Conventions

This section covers the typographical and syntax conventions used in this book.

Typographical Conventions

The SAS/C books for AmigaDOS use special fonts to depict specific types of information. These typographical conventions are as follows:

- roman* is the basic type style used for most text.
- monospace** is used to show example code. Monospace is used also for items specific to the C language, such as the names of functions, header files, and keywords.
- oblique* is used for arguments or variables whose values are supplied by the user; for example, you should enter an appropriate filename when you see *filename*.
- italic* is used for terms that are defined.

Syntax Conventions

This book uses the following conventions for syntax:

- monospace** indicates commands, keywords, and switches that should be spelled exactly as shown. These arguments may or may not be optional, depending on whether they are enclosed in square brackets.
- italic* indicates arguments for which you supply a value.
- [] (square brackets) indicate an optional argument when they surround the argument.
- . . . (ellipsis) indicates that you can repeat the argument or group of arguments preceding the ellipsis any number of times.
- | (vertical bar) means to choose one item from a group of items separated by the bars.

The following example illustrates these syntax conventions:

env [*function* | *integer*]

env

is a command name, so it appears in monospace type.

function

is a function for which you supply the name, so it appears in italic type.

[*function* | *integer*]

are both optional, so they are enclosed in square brackets.

function | *integer*

indicates that you can specify only one of the items separated by the vertical bar.

Additional Documentation

The following sections list documentation you may find helpful in using the SAS/C Development System.

SAS Documentation

If you are interested in SAS documentation, you need to contact Book Sales by writing to the following address or by calling the following number:

SAS Institute Inc.
Book Sales Department
SAS Campus Drive
Cary, NC 27513
919-677-8000

SAS/C Development System

This book is part of a set of publications for the SAS/C Development System. There are three other publications in the set:

- ☐ *SAS/C Development System User's Guide, Volume I: Introduction, Compiler, Editor, Version 6.0* describes how to
 - ☐ install the SAS/C Development System on your Amiga
 - ☐ use your development environment
 - ☐ create files in the editor
 - ☐ compile and link C files.
- ☐ *SAS/C Development System User's Guide, Volume II: Debugger, Utilities, Assembler, Version 6.0* describes how to use
 - ☐ CodeProbe, the SAS/C Source Level Debugger
 - ☐ utilities such as **smake**, **grep**, **scopts**, and **scmsg**
 - ☐ the SAS/C Macro Assembler
 - ☐ the global and peephole optimizers.
- ☐ *SAS/C Development System Quick Reference, Version 6.0* contains reference tables for the library functions, compiler and linker options, and debugger commands.

These volumes are sold together as a set. You can order them by specifying order #A56185.

Other Documentation

The following sections list additional reference material specific to the C language, Amiga and AmigaDOS programming, and the Motorola 68xxx microprocessor.

C Language

The following books describe the C programming language:

- American National Standards Committee (1990), *American National Standard for Information Systems—Programming Language C*, Document Number X3J11/90-013, Washington, DC: X3 Secretariat: Computer and Business Equipment Manufacturers Association.
- Harbison, Samuel P. and Steele, Guy L., Jr. (1990), *C: A Reference Manual, Third Edition*, Englewood Cliffs, NJ: Prentice-Hall, Inc.
- Kernighan, Brian W. and Ritchie, Dennis M. (1988), *The C Programming Language, Second Edition*, Englewood Cliffs, NJ: Prentice-Hall, Inc.

Amiga and AmigaDOS

The following books provide information specifically about programming on the Amiga. Most of the examples in these books are written in C or assembler.

- Commodore-Amiga, Inc. (1991), *Amiga Hardware Reference Manual, 3rd Edition*, Reading, MA: Addison-Wesley Publishing Company.
- Commodore-Amiga, Inc. (1991), *Amiga ROM Kernal Reference Manual: Devices, 3rd Edition*, Reading, MA: Addison-Wesley Publishing Company.
- Commodore-Amiga, Inc. (1991), *Amiga ROM Kernal Reference Manual: Includes and Autodocs, 3rd Edition*, Reading, MA: Addison-Wesley Publishing Company.
- Commodore-Amiga, Inc. (1991), *Amiga User Interface Style Guide*, Reading, MA: Addison-Wesley Publishing Company.
- Commodore-Amiga, Inc. (1991), *The AmigaDOS Manual, 3rd Edition*, New York, NY: Bantam Books.
- Commodore-Amiga, Inc. (1992), *Amiga ROM Kernal Reference Manual: Libraries, 3rd Edition*, Reading, MA: Addison-Wesley Publishing Company.

Motorola 68xxx

The following books contain information specifically about programming the Motorola 68xxx microprocessor:

- Motorola, Inc. (1987), *MC68881/MC68882 Floating-Point Coprocessor User's Manual, First Edition*, Englewood Cliffs, NJ: Prentice-Hall, Inc.
- Motorola, Inc. (1989), *MC68030 Enhanced 32-Bit Microprocessor User's Manual, Second Edition*, Englewood Cliffs, NJ: Prentice-Hall, Inc.
- Motorola, Inc. (1989), *Programmer's Reference Manual*, Phoenix, AZ: Motorola Literature Distribution.

Contacting CATS You can contact Commodore Application and Technical Support at the following address:

1200 Wilson Drive
West Chester, PA 19380
215-429-0643

1 What Libraries Can I Access?

- 1 *Introduction*
- 1 *Using the Shared Libraries*
- 2 *Using the Linkable Libraries*
- 3 *Creating Your Own Libraries*

Introduction

The SAS/C Development System includes several libraries. These libraries are of two types:

- shared libraries* are provided by Commodore. Many of the shared libraries are included with the standard WorkBench disk and are in the `libs:` directory. These libraries have the `.library` filename extension.
- linkable libraries* are included with the SAS/C Development System, for example `sc.lib` and `scm.lib`. These libraries are in the `lib:` directory and have the `.lib` extension.

The following sections describe each of the libraries supplied with your SAS/C software.

Using the Shared Libraries

Commodore provides many shared libraries, for example, `version.library`, `diskfont.library`, and `translator.library`. The autodocs available from Commodore describe the functions in these libraries. The autodocs are online text files, and you can purchase them from Commodore Amiga Technical Support (CATS).

You can access the Commodore shared libraries by one of the following methods:

- using the stub routines in **amiga.lib**. For more information, see “Using **amiga.lib**” in Chapter 2, “Using the Commodore Libraries.”
- including the correct prototype file (for example, **proto/dos.h**) in your program. The prototype files contain prototypes and **#pragma** statements for each of the functions included in **amiga.lib**. “Using the Prototype Files” in Chapter 2 describes how to use the **#pragma** statements.

Using the Linkable Libraries

Your SAS/C software contains several libraries of standard ANSI C functions, additional functions that are extensions to the ANSI standard, and functions that allow you to take advantage of your specific hardware configuration.

The SAS/C software provides two standard libraries:*

sc.lib is the SAS/C standard C library that contains all of the nonfloating-point routines described in Chapter 7, “Library Reference.”

scm.lib is the SAS/C standard C math library that contains all of the floating-point routines described in Chapter 7.

The remaining libraries contain the same routines as the standard libraries, but are for use with various compiler options. The following libraries contain special versions of routines found in **sc.lib**.*

scnb.lib is for use with far addressing (absolute data addressing).

scs.lib is for use with 16-bit integers.

scsnb.lib is for use with 16-bit integers and far addressing.

* The standard libraries now provide support for registerized parameters. To use registerized parameters, you must compile with the **PARMS=r** compiler option, but you do not need to link with a special library.

The following libraries contain special versions of routines found in `scm.lib`.*

- `scmieee.lib` is the SAS/C math library for use with the Commodore IEEE shared library.
- `scmffp.lib` is the SAS/C math library for use with the Commodore Motorola Fast Floating Point shared library.
- `scms.lib` is the SAS/C math library for use with 16-bit integers.
- `scm881.lib` is the SAS/C 68881 coprocessor math library.

Chapter 7 describes the functions contained in these libraries. Chapter 3, “Using the SAS/C Libraries,” describes how to access the linkable libraries.

Creating Your Own Libraries

You can also create your own shared or linkable libraries. For more information, see “Creating and Using Your Own Libraries” in Chapter 5, “Advanced Library Techniques.”

* The standard libraries now provide support for registerized parameters. To use registerized parameters, you must compile with the `PARMS=r` compiler option, but you do not need to link with a special library.

2 Using the Commodore[®] Libraries

- 5 *Introduction*
- 5 *Using amiga.lib*
- 6 *Using the Prototype Files*
- 7 *Defining Library Bases*

Introduction

Normally, when you call a C language function, you push the function parameters onto the stack. However, Commodore shared libraries expect to be called with parameters in specific registers, as described in the function description (`.fcd`) files from Commodore, and the library base in address register six (A6). You can solve this problem by using one of the following methods to access library functions:

- using the stub routines in `amiga.lib`
- using the `#pragma` statements in the prototype files.

This chapter describes how to access the Commodore libraries using these methods and how to define library bases for the Commodore libraries.

Using amiga.lib

`amiga.lib` contains all the stub routines necessary to call the Commodore shared libraries. These stub routines take the parameters off the stack, put them into the correct registers, load the correct library base, and call the routine.

For example, the following code calls the `Write` and `Output` functions in `dos.library` using the stub routines in `amiga.lib`. (The startup code opens the library, so you do not need to call the `OpenLibrary` function.)

```
void main(void)
{
    Write(Output(), "Hello World\n", 12);
}
```

If this program is in a file named `test.c`, you can compile, link, and run `test.c` with the following commands:

```
sc LINK test.c
test
```

The compiler, `sc`, produces an object file, `test.o`, and a link file `test.lnk`. `test.lnk` is an ASCII text file that tells the linker, `slink`, which libraries to use when linking. (`test.lnk` is called a `with` file.) The compiler options determine which libraries are specified in the `with` file. You can display `test.lnk` from a CLI (or shell) by entering `type test.lnk`.

In this example, two libraries will be searched: `sc.lib` and `amiga.lib`. The calls to the `Write` and `Output` functions will be resolved using the stub routines in `amiga.lib`.

Using the Prototype Files

Instead of using the stub routines in `amiga.lib`, you can use `#pragma` statements for the system libraries. The prototype files in the `include:proto` directory include prototypes and `#pragma` statements for each of the functions included in `amiga.lib`.

To use these statements, include the correct header file from the `include:proto` directory in your program. When you use these `#pragma` statements, the compiler can move function parameters into the correct registers, load the library base into register A6, and call the function directly, without the overhead required to use the stub routine.

For the example described in “Using `amiga.lib`,” you would add the following line at the top of the `test.c` file:

```
#include <proto/dos.h>
```

You can compile, link, and run `test.c` as before. Although the `test.lnk` file still lists `amiga.lib` as a library to be searched, the `Write` and `Output` functions are not pulled from `amiga.lib` because the compiler does not generate a symbol reference to those functions. From the `#pragma` statements, the compiler already knows where the routines are and which registers they require.

You can include all of the header files for the Amiga system libraries by including the following line in your program:

```
#include <proto/all.h>
```

Including `proto/all.h` increases the compilation time required for your program.

Defining Library Bases

The most common type of library base is a global variable. However, you can declare the base to be anything you want. For example, you can declare the base as automatic or static, or you can enter a complex `#define` statement such as

```
#define DOSBase mystruct->myDOSBase
```

You must include the `#define` statement before any reference to the base.

Also, before calling the functions in a library, you must initialize the library base. With the exception of `exec.library`, if the library file has the extension `.library`, initialize the base by calling the `OpenLibrary` function.

```
MyLibBase = OpenLibrary("name.library", revision);
```

If the library file has the extension `.resource`, initialize the base by calling the `OpenResource` function.

```
MyLibBase = OpenResource("name.resource", revision);
```

The *revision* number indicates the version of the operating system that you want to use.

OS Version	Library Revision
1.2	33
1.3	34
2.0	36 (Preliminary 2.0)
2.04	37

If the library file has the extension `.device`, initialize the base as follows:

```
OpenDevice("name.device",unit,&ioreq,flags);
MyLibBase=&ioreq.io_Device;
```

The *unit* and *flags* are specific to the device. Refer to the autodocs available from Commodore for more information.

The following tables list each of the Commodore shared library names, their library bases, and the header files in the `include:proto` directory that contain the `#pragma` statements and prototypes for the functions in those files.

Note: Case is significant in the library base names.

Table 2.1 Library Bases for `.library` Files

Name*	Base	Header File
<code>asl (2.0)</code>	<code>AslBase</code>	<code>asl.h</code>
<code>commodities (2.0)</code>	<code>CxBase</code>	<code>commodities.h</code>
<code>diskfont</code>	<code>DiskfontBase</code>	<code>diskfont.h</code>
<code>dos</code>	<code>DOSBase</code>	<code>dos.h</code>
<code>exec</code>	<code>SysBase</code>	<code>exec.h</code>
<code>expansion</code>	<code>ExpansionBase</code>	<code>expansion.h</code>
<code>gadtools (2.0)</code>	<code>GadToolsBase</code>	<code>gadtools.h</code>

(continued)

*Each of the library names listed in this table end with `.library`. For example, the full name of the `intuition` library is `intuition.library`.

Table 2.1 (continued)

Name*	Base	Header File
graphics	GfxBase	graphics.h
icon	IconBase	icon.h
iffparse	IFFParseBase	iffparse.h
intuition	IntuitionBase	intuition.h
keymap	KeymapBase	keymap.h
layers	LayersBase	layers.h
mathffp	MathBase	mathffp.h
mathieeedoubbas	MathIeeeDoubBasBase	mathieeedoubbas.h
mathieeedoubtrans	MathIeeeDoubTransBase	mathieeedoubtrans.h
mathieeesingbas	MathIeeeSingBasBase	mathieeesingbas.h
mathieeesingtrans	MathIeeeSingTransBase	mathieeesingtrans.h
mathtrans	MathTransBase	mathtrans.h
translator	TranslatorBase	translator.h
utility	UtilityBase	utility.h
workbench	WorkbenchBase	wb.h

*Each of the library names listed in this table end with .library. For example, the full name of the intuition library is intuition.library.

Table 2.2
Library Bases for
.device Files

Name	Base	Header File
console.device	ConsoleDevice	console.h
input.device	InputBase	input.h
ramdrive.device	RamdriveDevice	ramdrive.h
timer.device	TimerBase	timer.h

Table 2.3
*Library Bases for
.resource Files*

Name	Base	Header File
battclock.resource	BattClockBase	battclock.h
battmem.resource	BattMemBase	battmem.h
ciaa.resource	CiaBase	cia.h
ciab.resource	CiaBase	cia.h
misc.resource	MiscBase	misc.h
potgo.resource	PotgoBase	potgo.h

3 Using the SAS/C® Libraries

- 11 *Introduction*
- 11 *Determining Which Library to Use*
- 12 *Compiling with the Correct Library*
- 13 *Linking with the Correct Libraries*
- 15 *Using the Math Libraries*

Introduction

As described in Chapter 1, “What Libraries Can I Access?,” the SAS/C Development System includes several libraries. The names of these libraries indicate the type of routines in the library and, therefore, the compiler options you must use with the libraries. The following sections describe the naming conventions for SAS/C libraries and the compiler options to use with each library.

Determining Which Library to Use

If you know what type of library routines you need to use, you can determine the library you need to use by looking at the letters included in the library name. The names of the SAS/C libraries follow this format:

`sc[s][nb][m[ieee | ffp | 881]].lib`

where

- s** indicates the library uses short integers.
- nb** indicates the library uses far addressing (no base register).
- m** indicates the library uses floating-point math. If the library is a math library, the library name extension indicates the type of floating-point routines supported by the library, as follows:
 - none** `scm.lib` contains standard SAS/C floating-point routines.

ieee The library contains Commodore IEEE routines.

ffp The library uses the Motorola fast floating-point format.

881 The library uses Motorola 68881 instructions.

Note: Each type of math library is mutually exclusive. You can use only one math library at a time.

For example, **scnb.lib** uses far addressing, **scs.lib** uses 16-bit integers, and **scsnb.lib** uses far addressing and 16-bit integers.

Because of disk space considerations, all possible libraries are not included in this software. The libraries supplied in this software are described under “Using the Linkable Libraries” in Chapter 1.

Because not all possible libraries are included, some combinations of compiler options are not supported. For example, combining the use of short integers with 68881 instructions would require a library named **scms881.lib**, which is not supplied with this software. Therefore, the combination of compiler options **shortint** and **math=68881** is not supported. In addition, the combination of options **shortint** and **math=ieee** (**scmsieee.lib**) is not supported.

If you need a library that is not provided with the SAS/C Development System, contact the Technical Support Division at SAS Institute.

Compiling with the Correct Library

Certain compiler options tell the compiler which SAS/C library to use. The compiler chooses the correct libraries based on the option or combination of options you specify.

The following lists show the compiler options you should specify for the types of library routines you need to use.

parms=register

tells the compiler to use registerized parameters.*

shortint

tells the compiler to use short integers.

data=far

tells the compiler to use absolute data addressing (no base register).

* The standard libraries now provide support for registerized parameters. To use registerized parameters, you must compile with the **parms=register** compiler option, but you do not need to link with a special library.

The **math=** options are mutually exclusive and indicate the type of floating-point math to be used, as follows:

math=standard

tells the compiler to use standard SAS/C floating-point routines.

math=ieee

tells the compiler to use Commodore IEEE routines.

math=ffp

tells the compiler to use the Motorola fast floating-point format.

math=68881

tells the compiler to use Motorola 68881 instructions.

Linking with the Correct Libraries

You can run the compiler and linker (**slink**) as separate steps (call each tool explicitly), or you can specify the **link** option in the **sc** command to link your program automatically after it compiles.

If you call **slink** separately, you must specify each library explicitly after the **lib** keyword. Make sure you link your libraries in the correct order. When you are linking more than one library, follow these rules:

1. Specify any libraries you create first.
2. Specify your math library.
3. Specify any additional nonmath libraries.
4. Specify **amiga.lib** last.

For example, to compile and link **prog.c** using short integers, you could call the compiler and linker separately as follows:

```
sc shortint prog.c
slink lib:c.o+prog.o to test lib lib:scs.lib lib:amiga.lib
```

If you want to compile and link your program with one command, enter the **link** option in addition to any other options you need on the **sc** command line. If the compiler does not find any errors in your program, it will look at the other options you specify, determine which libraries are needed, and call **slink** to link your program. The compiler will link the libraries in the correct order. The compiler uses the standard C library **sc.lib** if you specify **link** without any additional library options. If you specify **math=standard** and **link** without any additional library options, the compiler uses **sc.lib** and the math library **scm.lib**.

For example, to compile and link `prog.c` with one command, use the `link` compiler option as follows:

```
sc shortint link prog.c
```

To compile and link a program using the standard C library, you can use the following command:

```
sc link prog.c
```

If `prog.c` uses floating-point math and you want to use only the standard floating-point library (`scm.lib`), specify the `math=standard` option also:

```
sc math=standard link prog.c
```

To summarize, Table 3.1 shows the options you should use on the `sc` command line to compile and link with each library supplied with the SAS/C Development System.

Table 3.1
*Options Required to
Compile and Link with
Each Library*

Library	Options	Type of Library
<code>sc.lib</code>	<code>link</code>	Standard C library
<code>scs.lib</code>	<code>shortint</code> <code>link</code>	Version of <code>sc.lib</code> that uses 16-bit integers
<code>scnb.lib</code>	<code>data=far</code> <code>link</code>	Version of <code>sc.lib</code> that uses far addressing
<code>scsnb.lib</code>	<code>shortint</code> <code>data=far</code> <code>link</code>	Version of <code>sc.lib</code> that uses 16-bit integers and far addressing
<code>scm.lib</code>	<code>math=standard</code> <code>link</code>	Standard math library
<code>scmieee.lib</code>	<code>math=ieee</code> <code>link</code>	IEEE math library supplied by Commodore

(continued)

Table 3.1
(continued)

Library	Options	Type of Library
<code>scmffp.lib</code>	<code>math=ffp</code> <code>link</code>	Motorola fast floating-point math library
<code>scms.lib</code>	<code>math=standard</code> <code>shortint</code> <code>link</code>	Math library for use with 16-bit integers
<code>scm881.lib</code>	<code>math=68881</code> <code>link</code>	68881 coprocessor math library

If you are porting code from a machine with short (16-bit) integers, you should compile with the `shortint` option. The `shortint` option tells the compiler to use 16-bit integers instead of the normal 32 bits. Any code compiled with the `shortint` option must be linked with a library whose name contains the letter `s` after the `sc`.

It is recommended that you use the `far` keyword in the declarations and definitions of some of your larger data structures to move them into the far data section. Continue adding the `far` keyword until the near data section is less than 64K. Alternatively, if your program has no large data structure, you may want to use `data=far`, signifying long (32-bit) references, to force all data into the far section. For more information on near and far data sections, refer to Chapter 12, “How Does the Compiler Work?” in the *SAS/C Development System User’s Guide, Volume I: Introduction, Compiler, Editor*.

Using the Math Libraries

The standard math library, `scm.lib`, contains all of the floating-point routines described in Chapter 7, “Library Reference.” The other math libraries contain special versions of the routines in `scm.lib`.

The following list contains additional information about using the math libraries.

`scm.lib`

is the standard math library. You must specify a math library if your program performs floating point operations. To use this library, specify the `math=standard` option on the compiler command line. This library ignores any math coprocessors that may be present.

scmieee.lib

calls the IEEE shared math libraries supplied by Commodore to compute floating-point operations. Advantages of using this library include:

- smaller executable code size. Code size is reduced because much of the code needed to perform the floating-point operations is in the shared library.
- optional 68881 or 68882 support. The Commodore IEEE library uses the 68881 or 68882 chip, if it is present, thereby making your program run much faster. If a math coprocessor chip is not present, your program will still run.

The Commodore routines are slightly slower than their counterparts in **scm.lib** if the 68881 coprocessor is not present.

scmffp.lib

performs its operations in Motorola Fast Floating-Point (FFP) format. The FFP routines are faster than the IEEE routines, but they obtain this speed at the expense of accuracy. The precision of an FFP double is half that of an IEEE double. However, for many applications, FFP accuracy is sufficient.

The compiler manipulates real numbers in the FFP format if you specify the **math=ffp** option. Do not mix the FFP and IEEE formats. If one module in an executable uses the **math=ffp** option, then all modules must use it.

This library calls **mathffp.library** and **mathtrans.library**, which are supplied by Commodore.

scms.lib

is the same as the standard math library except it uses 16-bit integers.

scm881.lib

is the SAS/C 68881 Coprocessor Math Library. To get the best performance from this library, include the **m68881.h** header file (**#include <m68881.h>**) and compile with the **math=68881** option. Programs linked with this library will not run on a machine without a 68881 coprocessor.

If you call floating-point math routines in your program, but you have not compiled or linked with the correct options, you will get messages such as:

```
Undefined Symbol: __CXnnn
```

To correct the problem, specify the correct compiler option for the library you need: **math=standard**, **math=ieee**, **math=ffp**, or

math=68881, as described earlier in this chapter. If you want the compiler to call **slink** automatically to link your program, you must also specify the **link** option.

4 Using the System Header Files to Access Libraries

- 19 *Introduction*
- 19 *Compressing Header Files*
- 20 *Using Built-in Functions*
- 21 *Using Global Symbol Tables*
 - 21 *Creating a GST*
 - 22 *Header File Restrictions*

Introduction

The system header files contain all of the function prototypes and `#define` statements needed by the libraries. This chapter describes how to use header files to reduce the amount of code produced by your program and the time required to compile your program. This chapter describes how to do the following:

- compress header files
- use the built-in functions in the `string.h` and `stdlib.h` files
- use Global Symbol Tables (GSTs).

Compressing Header Files

The compiler can read compressed header files faster than uncompressed header files. By compressing header files, you can increase the overall performance of your compiler without sacrificing ANSI compliance. Compressed headers require less disk space than uncompressed headers.

The compiler can read header files in both compressed and uncompressed forms. You handle each form the same way; you do not have to specify anything special on your command line or in an `#include` statement. Also, you can have compressed and uncompressed header files within the same directory.

To compress a header or source file, enter the `scompact` command as follows:

```
scompact source-file destination-file
```

To compress all header files in a directory, enter the source file directory as the *source-file* and the destination directory as the *destination-file*. To compress all header files in a directory and its subdirectories, append the **ALL** keyword, as follows:

```
scompact source-directory destination-directory ALL
```

The utility encodes commonly occurring Amiga keywords.

Using Built-in Functions

Built-in functions, which are encouraged by the ANSI Standard, allow the compiler to exploit register contents. When you use built-in functions, the compiler produces less code than if you called the library function. For example, if you use the built-in function **strlen**, the compiler does not generate code if the input string is a constant string.

Also, a built-in function that returns a pointer will not return the pointer if the pointer is unused.

Your SAS/C software contains the following built-in functions:

abs	getreg	memset	strcmp
emit	memcmp	printf	strcpy
geta4	memcpy	putreg	strlen

When your program calls the **printf** function, the compiler examines the formatting string, and takes one of the following actions:

- If the formatting string is a constant string with no substitutions, the compiler changes the **printf** call to a **_writes** call.
- If the formatting string is a constant string and contains only **%s**, **%d**, and **%p** formats, the compiler calls the **_tinyprintf** function. **_tinyprintf** is a private function that you should not call directly. The compiler will call **_tinyprintf** if you include the **stdio.h** file.
- If double-precision floating-point or floating-point numbers are present, the compiler calls the **__mprintf** function.

The **#define** statements in the header files **string.h** and **stdlib.h** are set to call the built-in functions (except for **printf**, which is in **stdio.h**). To use these functions, you must include the correct header file. If you do not want to use the built-in functions, you can modify the appropriate function with an **#undef** statement. For example, if your program contains an **#undef strcmp** statement, **strcmp** will be accessed as a real function call, not as a built-in function.

Using Global Symbol Tables

A *Global Symbol Table* (GST) is a collection of all the information defined in a set of header files, stored in a format that the compiler can use easily and quickly. You can use GSTs to reduce significantly the processing time required for header files.

In a typical development environment, processing header (include) files takes far more time than processing the source file itself. With a complete GST for your project, the compiler reads the information in the header files only once and stores it in memory. When you include a header in your program, the compiler looks in the GST before it searches any of the directories in the normal search path. If the header information is already in the GST, the compiler does not process the header file. If a header file is included more than once, the compiler does not read the header file again.

Previous versions of the SAS/C Development System include precompiled header files. GSTs are an improved version of precompiled header files.

Creating a GST

When you start a new project, no GST exists for the project. You can create a GST manually by specifying which header files you want included in the GST, or the compiler can generate one automatically for you. Once the information from the header files is in the GST, the header file need not be present to compile the source file.

To generate a GST, compile your program with the **makegst** compiler option, as follows:

```
sc makegst filename program
```

You can use any filename for the GST. A consistent naming convention will help you recognize GSTs. For example, you can compile **header.c** and use the filename **header.gst** by entering the following command:

```
sc makegst header.gst header.c
```

When the compiler creates a GST file, only the data in your header files and system header files are included in the GST. Data in the C source file are ignored. Therefore, you can use a normal project C source file to generate a GST. (In previous versions of the SAS/C Development System, you need a separate C source file to generate a precompiled header file.)

Any objects declared in a GST are assumed to be external variables (**extern**). Therefore, these objects must be declared in one of the C modules to be linked into the executable file.

To use a GST, use the **gst** option on the **sc** command line. You can specify the **gst** option only once per compilation. For example, if your

GST is named `project.gst`, you can compile `project.c` by entering the following command:

```
sc gst=project.gst project.c
```

For more information on creating and managing GSTs, refer to the description of the GST utility in Part 2, “Using SAS/C Utilities,” of the *SAS/C Development System User’s Guide, Volume II*.

Header File Restrictions

There are some restrictions on the contents of the header files that can be included in a GST. If a header file does not meet these restrictions, you must exclude it from the GST as described in the *SAS/C Development System User’s Guide, Volume II*. These restrictions are as follows:

- Header files that you want to include in a GST cannot define data items or functions. You can declare a data item and include function prototypes, but you cannot define data items or functions.

The difference between a declaration and a definition is that a definition actually allocates storage for a variable, but a declaration only informs the compiler of the variable’s type. For example, the statement `int x;` defines an integer data item called `x`. The statement `extern int x;` declares `x` but does not define it.

If the header file included in the GST contains a data definition, the data definition is treated as if it were a declaration. Storage is not allocated for the item, and unless it is defined elsewhere, the item shows up at link time as undefined. Any static initializer expressions generate an error message.

- Header files that you want to include in a GST must define the same symbols to the same values no matter which source file in the project includes the GST.

A GST is precompiled, so you cannot change what symbols it defines by changing preprocessor symbols at compile time. For example, the following file defines different symbols if the preprocessor symbol `MAIN_C` is defined:

```
#ifdef MAIN_C
#define x 1
#else
#define x 2
#endif
```

The previous lines will not work in a GST file. If the symbol `MAIN_C` was defined when the header was added to the GST, then `x`

will be 1; if not, **x** will be 2. In both cases, **x** will be the same value for all source files in the project, regardless of whether **MAIN_C** was defined for that source file.

5 Advanced Library Techniques

25	<i>Introduction</i>
25	<i>Using #pragma Statements</i>
26	<i>Creating Your Own #pragma Statements</i>
27	<i>Calculating the Magic Value</i>
29	<i>Generating #pragma Statements with the fd2pragma Utility</i>
31	<i>Creating and Using Your Own Libraries</i>
31	<i>Custom Linkable Libraries (Link at Link Time)</i>
33	<i>Custom Shared Libraries (Load at Run Time)</i>

Introduction

As described in Chapter 2, “Using the Commodore Libraries,” you can access the Commodore shared libraries using stub routines or using `#pragma` statements. You can also access other shared libraries using stub routines and `#pragma` statements, and you can create your own libraries.

This chapter describes how to do the following:

- create your own `#pragma` statements
- use the `fd2pragma` utility to create `#pragma` statements
- create your own shared or linked libraries.

Using #pragma Statements

When you access functions with a `#pragma` statement, the compiler moves library function parameters into the correct registers, loads the library base into register A6, and calls the function. If you do not use `#pragma` statements, the compiler pushes the function parameters onto the stack, calls the stub routine which takes the parameters off the stack, places them into the correct registers, and then calls the function.

Using `#pragma` statements reduces program run time and saves development time (because you save time linking your program). However, the code generated by this technique does not usually show any significant savings in size over code generated using stub routines.

You can write your own `#pragma` statements, or you can use the `fd2pragma` utility to generate the `#pragma` statements.

Creating Your Own #pragma Statements

Your SAS/C software supports four types of **#pragma** statements:

- libcall** allows you to call any library routine. This statement passes all parameters through specified registers.
- syscall** allows you to call a routine in **exec.library**. This statement passes all parameters through specified registers.
- tagcall** allows you to call any library routine and pass parameters through registers and the stack.
- msg** allows you to enable and disable errors and warnings. For more information, refer to the *SAS/C Development System User's Guide, Volume I: Introduction, Compiler, Editor*.

The **libcall** statement has the following format:

```
#pragma libcall base routine offset magic
```

where

- #pragma** identifies the statement.
- libcall** indicates that you are accessing a library.
- base** specifies the pointer to the library base.
- routine** identifies the library routine you want to call.
- offset** specifies the offset from the library base of the routine in hexadecimal format. You can get offset values from the function description (**.fd**) files.* For example, the offset value for the **AllocMem** function in **exec.lib** is C6 (decimal 198).
- Note:** The offsets are subtracted from, not added to, the library base.
- magic** describes the register conventions expected by the routine. The following section, "Calculating the Magic Value," describes how to determine magic values.

For example, to call the **Open** function in **dos.library**, enter the following statement:

```
#pragma libcall DOSBase Open 1e 2102
```

If the library you want to access is **exec.library**, you can use the **syscall** form of the **#pragma** statement:

```
#pragma syscall routine offset magic
```

* To make sure you get an accurate value, check the files on your disks instead of relying on the published listing of those files in the AmigaDOS manuals.

The `#pragma syscall` statement is roughly equivalent to a `#pragma libcall` statement using `SysBase` as the library base:

```
#pragma libcall SysBase routine offset magic
```

The `#pragma tagcall` statement allows you to pass a varying number of parameters to the routine because some parameters are passed through the stack. The `#pragma tagcall` statement follows the same format as the `#pragma libcall` statement; however, the compiler uses the information you specify in the `magic` value differently. The following section, “Calculating the Magic Value,” describes the differences.

Calculating the Magic Value

The format of the `magic` number is as follows:

```
nmlkjihgfedcbaR#
```

The lowercase letters a through n are hexadecimal numbers that specify registers corresponding to the parameters in *right to left order*. For the `#pragma libcall` and `#pragma syscall` statements, the hexadecimal numbers designate the registers into which the first through the fourteenth function parameters should be placed.

For the `#pragma tagcall` statement, each hexadecimal number you specify, except the last, designates a register into which a parameter should be placed. The last number designates a register that contains a pointer to all parameters that are not placed into registers. (These parameters are placed on the stack and referred to as the *taglist*.)

The following table lists hexadecimal numbers you should use to designate specific registers.

Table 5.1
Register Conventions

Value (Hex)	Register	Value (Hex)	Register
0	D0	7	D7
1	D1	8	A0
2	D2	9	A1
3	D3	A	A2
4	D4	B	A3
5	D5	C	A4
6	D6	D	A5

Note: Address registers A6 and A7 are never used as part of a **#pragma** statement. Register A6 is used as the library base for library function calls, and register A7 is the stack pointer.

The uppercase *R* specifies the register where the result is returned. This number is usually 0 for data register D0, although there are no restrictions as to which register you may select for return values.

The **#** value specifies the total number of parameters passed to the routine in registers. You must specify this value in hexadecimal, and it can be up to 14 (a hexadecimal E).

For example, refer to the synopsis of the AmigaDOS **Open** function in the *Amiga ROM Kernel Reference Manual: Libraries*.

```
file = Open(name, accessMode)
D0          D1          D2
```

Table 5.1 gives the hexadecimal value for register D0 as 0, the value for register D1 as 1, and the value for register D2 as 2. The **Open** function takes two parameters and returns the result in register D0. Therefore, the magic value for the call to the **Open** function is 2102.

As an additional example, refer to the synopsis of the AmigaDOS **OpenLibrary** function in the *Amiga ROM Kernel Reference Manual: Libraries*.

```
library = OpenLibrary(libname, version)
D0          A1          D0
```

Using the information in the following table, you can calculate the magic value for the **OpenLibrary** function as 0902:

Magic Number	Meaning
0	Register for version (D0)
9	Register for libname (A1)
0	Register for the result, library (D0)
2	Number of parameters

To illustrate the **#pragma tagcall** statement, suppose you have a function called **myfunc** that can take a variable number of parameters.

Suppose the function `myfunc` has two parameters followed by any optional taglist parameters. You can use `myfunc` with a `#pragma tagcall` statement with a magic value of 98103. The first parameter would be placed into register D1, the second into register A0, all remaining parameters would be placed on the stack, and the address of the first parameter on the stack would be placed into register A1.

Generating #pragma Statements with the fd2pragma Utility

The format of the `#pragma` statements is relatively easy to understand and code but difficult to maintain and or decode. However, you can create a standard ASCII function description (`.fd`) file, and use the `fd2pragma` utility to create a header file containing the corresponding `#pragma` statements.

Creating a function description (.fd) file

A function description file consists of instructions (*directives*) to the `fd2pragma` utility, the entry points of each library function, and comments. You can enter only one statement per line. Comments begin with an asterisk (*); instructions to the `fd2pragma` utility begin with two pound signs (##); any other line is an entry point into a library function.

For example, the following lines are from the `dos.lib.fd` file:

```
* "dos.library"
##base _DOSBase
##bias 30
##public
Open(name,accessMode)(d1/d2)
Close(file)(d1)
Read(file,buffer,length)(d1/d2/d3)
*
##end
```

The `fd2pragma` utility accepts the following instructions:

##base name	identifies the global variable pointing to the base of the library.
##bias n	identifies the offset value between the library base and the entry point of the first function in the library. Enter 30 for this value.
##public	identifies subsequent functions as being available to your programs.
##private	identifies subsequent functions as being available to the library only. Programs cannot call functions designated as private. ##private is required for

externally unavailable functions because the offsets are calculated from the positions of the function names in the lists in the `.fd` file. Skipping a private function in the list would cause incorrect offsets to be calculated for subsequent functions.

##end Marks the end of the file.

You must include entry points for all of the functions in your library. The entry points follow this format:

entryname(arguments)(registers)

Use commas to separate the arguments. Use commas or slashes (/) to separate registers.

Note: Commas are used to separate entries that need to be pushed with separate `move` or `movem` assembler instructions. Slashes indicate those entries that can be combined in a single `movem` instruction.

For example, the following lines are from the `intuition.library` file:

```
InitCode(startClass,version)(D0/D1)
PrintIText(rp,itext,left,top)(A0/A1,D0/D1)
```

Running the `fd2pragma` utility

To run the `fd2pragma` utility, enter the `fd2pragma` command as follows:

```
fd2pragma infile.fd newheader.h
```

where

infile specifies the function description file that you created. The `.fd` extension is required.

newheader specifies the file where you want the new `#pragma` statements stored. The `.h` extension is required.

If the `fd2pragma` utility encounters any errors, the errors are printed in the standard output file. For example, your function description file could contain invalid directives, invalid register specifications, or no `##end` statement.

Using the `fd2pragma` utility is less likely to lead to errors than trying to amend the `#pragma` statements themselves. It is strongly suggested that you always maintain the function description file and use the `fd2pragma` utility.

Creating and Using Your Own Libraries

A linkable library, in its simplest form, is a concatenation of object modules. An *object module* is the output of the compiler and cannot be run. Object modules are used by the linker, `slink`, to create the final executable module.

You can create your own linkable libraries using the Object Module Librarian (OML) or the AmigaDOS `JOIN` command. To use your custom linkable library, you must link with it before you run your program. If you have code that rarely changes and is used in multiple projects, you should create a linkable library.

You can create shared libraries using `slink` and the `fd2pragma` utility. To use your custom shared library, you do not need to link with it, but you must call the `OpenLibrary` function before you can access any of your library's routines. You should create a shared library if one or more of the following are true:

- ☐ you have code that must be shared among more than one program.
- ☐ you have a code layer that needs to do different things depending on run-time information.
- ☐ you have code that will rarely be used and you want to save system memory.

You should name your libraries with the `.lib` filename extension.

The following sections describe how to create and use your own linkable and shared libraries.

Custom Linkable Libraries (Link at Link Time)

You can create a custom linkable library with one command, either `JOIN` or `oml`. To use your new library, you must specify the library in the `sc` or `slink` command line.

When a function is included from a library, all other functions from the same source file are also included in your executable module. Therefore, you may want to place different functions in different files before creating your library. The following sections describe the steps necessary to create and use a linkable library.

Creating the library

To create linkable libraries, you can use the `JOIN` command as follows:

`JOIN object-files to library`

For example, you could create `lib1.lib` from three separate object files with the following command:

```
JOIN obj1.o obj2.o obj3.o to lib1.lib
```


The `.o` and `.lib` extensions are required.

Using the `JOIN` command has disadvantages. First, you cannot replace a single object module with another. Second, the linker cannot efficiently look up a symbol in this type of library because it has to scan all the object modules.

Using the Object Module Librarian (OML) produces a more efficient library. To create a new library with the OML, enter the `oml` command as follows:

```
oml library r ofiles
```

For example, to create the same library as shown in the `JOIN` command, you would enter the following `oml` command:

```
oml lib1.lib r obj1.o obj2.o obj3.o
```

You should give your library name the `.lib` file name extension.

The letter `r` is a command that tells the OML to replace the object files in the library or add them if they are not already present. The OML also supports additional commands and options. For a complete description of the OML, refer to the *SAS/C Development System User's Guide, Volume II: Debugger, Utilities, Assembler*.

The OML uses an indexing scheme that makes symbol lookup much faster for the linker. You can also replace single object modules without rebuilding the entire library.

Calling functions in the library

To use your new library, you must link with it. To link in your custom libraries, specify each library in the `sc` command, as follows:

```
library1.lib library2.lib
```

For example, `lib:mylib.lib df0:yourlib.lib` tells the linker to search the libraries `lib:mylib.lib` and `df0:yourlib.lib` before it searches the default libraries.

The linker also includes the basic run-time support library, `lib:sc.lib`, and the AmigaDOS binding library, `lib:amiga.lib`, after it includes the libraries you specify.

To compile a program called `prog.c` and link in the library created in the previous examples, `lib1.lib`, you would enter the following command:

```
sc link lib1.lib prog.c
```

The compiler puts `lib1.lib` in the list of libraries to be searched before the standard libraries. The `with` file that is passed to `slink` is as follows:

```
FROM lib:c.o prog.o
to prog
lib lib1.lib lib:sc.lib lib:amiga.lib
```

Custom Shared Libraries (Load at Run Time)

Creating a custom shared library is more complicated than creating a linkable library. You need to know how to create function description files and how to use `slink`.

You can create two types of shared libraries:

A library with one near data section

With this type, all tasks that open the library share the same data space. Extra care should be used when accessing the near data to avoid non-reentrant functions. (With previous releases of the SAS/C Development System, this type of library was the only type available.) To create this type of library, link with `libinit.o` as described in the following section.

A library that copies the near data section

With this type, each task that opens the library uses a separate near data section, but all openers share the same far data section. The library copies the near data section and passes the library base to the task that opens the library. That is, each call to the `OpenLibrary` function returns a different library base. (This technique is similar to the technique `cres.o` uses for resident programs.) To create this type of library, link with `libinitr.o` as described in the following section.

The following sections describe the steps necessary to create and use both types of shared libraries. `sc:examples/samplelib` contains an example of creating a shared library.

Creating the library

To create a shared library, follow these steps:

1. Decide if your library will have one near data section for all tasks or a separate section for each task.
2. Determine which functions will be in your library and the registers in which they take their parameters. Declare the function using the `__asm` keyword, and declare the parameters with the `register` keyword, two underscores, and the register into which you want the parameter placed. For more information, refer to “Specifying

Registers for Parameters” in Chapter 11, “Using Assembly Language with C,” in the *SAS/C Development System User’s Guide, Volume II*.

3. Create a function description file (`.fd`) describing your library using the information in “Creating a Function Description (`.fd`) File,” earlier in this chapter. The register information should match the function declarations.
4. Add the `__saveds` keyword to the function definition for any function that is callable from outside the library.
5. If any custom initialization code for the library is required, create a function with the following prototype in your source code:

```
int __saveds __UserLibInit(void);
```

Place any required initialization code, such as opening libraries or allocating memory, into this function. This function will be called by the initial startup code of the library if you are using one data section or by the `OpenLibrary` function if you are using separate data sections. If `__UserLibInit` is successful, it should return a 0; otherwise, it should return a nonzero value. If a nonzero value is returned, the startup code for the library aborts, and the library is not opened.

Any system resources allocated by the `__UserLibInit` routine must be freed. To do so, create a function with the following prototype in your source code:

```
void __saveds __UserLibCleanup(void);
```

Place all cleanup code, such as code to close libraries and free memory, into this function. The `__UserLibCleanup` function will be called by the `LibExpunge` function just before the library is removed from the system if you are using one data section or by the `CloseLibrary` function if you are using separate data sections.

6. Compile all modules with the `libcode` and `nostackcheck` compiler options.
7. Create the library using the `slink` command, as follows:

```
slink libfd fdfile [libprefix prefix]
[libid idstring]
to libname.library
from lib:libent.o lib-init-routine
modules
[lib libraries][libVersion version][libRevision revision]
```

where

fdfile

specifies the name of the function description file you created in step 3.

prefix

specifies a prefix you want added to the functions listed in the function description file. The default is an underscore (`_`).

Make sure that the function declarations in your source code match the full name, including any specified prefix. For example, if your library has a function called `myfunc` and you specify a prefix of `_LIB`, you should declare the function in your source code as `LIBmyfunc`.

idstring

specifies a library identification string. `slink` uses this value to set the `_LibID` symbol, but *idstring* is not required.

libname.library

specifies the resulting library name. You should give your library name the `.library` file name extension.

libent.o

contains the `RomTag` for the library and must be the first object module listed. `libent.a`, the source for `libent.o`, is in the `source` subdirectory.

lib-init-routine

contains the system-required library entry points `Open`, `Close`, and `Expunge`, as well as other initialization routines. Specify either `libinit.o` or `libinitr.o` as described at the beginning of this section. The library initialization routine must be the second object module listed. `libinit.c`, the source for `libinit.o`, is in the `source` subdirectory.

modules

specifies the modules that you want included in your library. They can be in any order as long as they are specified after `libent.o` and `libinit.o`.

libraries

specifies any libraries that you want the linker to search for modules referenced in your code.

version

sets the version number of the library you are creating. You can set the version number to any number from 0 to 255. The default is 1.

revision

specifies a minor revision number for your library. If you do not specify a revision number, it defaults to 0.

8. Create **#pragma** statements for your library using the **fd2pragma** utility. **fd2pragma** produces a header file containing **#pragma** statements for all the externally callable functions in your library. See “Generating **#pragma** Statements with the **fd2pragma** Utility,” earlier in this chapter, for information on using **fd2pragma**.

Calling functions in the library

To use the functions in the new library, follow these steps:

1. Include the header file generated by the **fd2pragma** utility in your program.
2. Declare a variable in your program of type **struct Library *libbase**. The library base can be an automatic or a global variable. The name of the library base is determined from the **##base** instruction in the function description file. For example, if your library is called **mylib.library**, and you named the library base **MyLibBase**, you would declare the library base as follows:

```
struct Library *MyLibBase;
```

3. Initialize the library base in your program by calling the **OpenLibrary** function.

```
MyLibBase = OpenLibrary("mylib.library", version);
```

The *version* number indicates the minimum version number of the library that you want to use. You can set the version numbers of any libraries that you create by specifying the **version** keyword in the **slink** command. A *version* of 0 will open any version. You must initialize the library base before calling any functions in the library, and you should check to make sure that the library base is not **NULL**. A null library base indicates that the library could not be loaded or that the version was not acceptable.

4. Call whatever library functions are needed. You can call these functions by name, just like any other function. When one of the library functions is called, the compiler loads the parameters into the correct registers, loads the library base in register A6, and calls the function.

5. Call the `CloseLibrary` function at the end of your program:

```
CloseLibrary(MyLibBase);
```

Restrictions on shared libraries

For a library with one near data section, all tasks that open the library get the same far data section and, if using `libinit.o` instead of `libinitr.o`, the same near data section. Therefore, if two programs open your library at once, you must not write to the library's global data in a way that will affect the other program.

To use DOS calls in your library, you must initialize `DOSBase` in your `__UserLibInit` routine. If you are using `#pragma libcall SysBase` to call `exec.library` routines, you must also initialize `SysBase`. The recommended method is to use `#pragma syscall`, which you can do by not defining the preprocessor symbol `__USE_SYSBASE`. This method is the default.

If you modify the provided `libinit.c` file, you cannot refer to global data in this file. You can call a function in another file that refers to global data. `slink` does not fix up global data references in the `libinit.c` file.

Understanding shared libraries

To help you build your own libraries, the following sections describe what happens when you create a library using the information in the earlier section "Custom Shared Libraries (Load at Run Time)." These sections explain how multiple processes can access the same shared library simultaneously.

What the compiler does

Modules compiled with the `libcode` option do not use register A6, except when calling other libraries, in which case the value of the register is restored immediately on return from the library. Therefore, register A6 always points to the current library base.

When you specify the `__saveds` keyword, the compiler generates the following instruction in the function prolog:

```
LEA    LinkerDB(A6),A4
```

`LinkerDB` is a symbol generated by the `slink` command that specifies the offset from the library base at which user global data may be stored. Therefore, the above instruction sets register A4 to point to the library's global data.

Note: The standard executable startup code, `c.o`, contains a reference to `LinkerDB`. When `slink` is linking a library, it interprets `LinkerDB` differently than when linking a normal program.

What the `LibInit` function does

The `LibInit` function, in a shareable library, sets up the library base. In addition, it copies global data from the place where the `LoadSeg` function loaded them (as defined by the symbol `__Libmergeddata`, which is generated by the `slink` command) to the end of the library base structure. After the global data are copied, any data relocations are performed. From this point on, global data can be referenced relative to register A4, since A4 has been set to point at the global data.

What the `slink` command does

First, `slink` reads the function description file, determines the names and number of functions to the library, and creates a jump table of the following form:

```
__LibOpen
__LibClose
__LibExpunge
RomTag - 2
user-defined functions
```

The names of the user-defined functions are created by adding the specified prefix (see step 4 under “Creating the Library” earlier in this chapter) to the front of the function names in the function description file. The symbol `__LibFuncTab` points to this jump table, which is placed into the data section.

`slink` communicates information to `libinit.o` by defining the following global symbols:

<code>__LibName</code>	is the name of the library, as specified by the <code>to</code> option to <code>slink</code> .
<code>__LibID</code>	is the library ID, as specified by the <code>libid</code> option to <code>slink</code> . <code>slink</code> sets this string to null if you do not specify a <code>libid</code> .
<code>RESLEN</code>	is the length of the data section to be allocated as part of the Library structure.
<code>NEWDATA</code>	is the length of the initialized data section.

The `libinit.o` file references these symbols at startup to correctly initialize the library and all associated data.

Understanding the format of shared libraries

The following lines show the format of the library that `slink` produces.

```
Code Hunk 1: (comes from libent.o)
    moveq    #0,d0          ; This is in case someone tries
    rts      ; to run the library as a program
RomTag
    dc.w     RTC_MATCHWORD   ; RomTag information for the library
    dc.l     _LibRomTag      ; as described in Rom Kernel manual
    dc.l     endtag          ; |
    dc.b     RTF_AUTOINIT    ; |
    dc.b     VERSION         ; |
    dc.b     NT_LIBRARY      ; |
    dc.b     PRI             ; |
    dc.l     _LibName        ; |
    dc.l     _LibID          ; v
    dc.l     _LibInitTab     ; End of RomTag information
```

```
Code Hunk 2 - n: (comes from libinit.o and user modules)
_LibInit
    <code for LibInit>
_LibOpen
    <code for LibOpen>
_LibClose
    <code for LibClose>
_LibExpunge
    <code for LibExpunge>
    user functions
```

DATA HUNK

```
_Libmergeddata    ; symbol that tells LibInit where the merged
                  ; data is initially loaded.
_LibInitTab        ; struct LibInitTab as specified in libinit.c
    length of global data
    pointer to function jump table
    NULL
    pointer to initialization code (_LibInit)
    user global data
```


6 Predefined Data Name Reference

This chapter describes the predefined data names provided with the SAS/C software. These data names are external variables that you can read or write from your program. You can also form a pointer to any external name using the ampersand (&) operator.

The data names are described in alphabetical order. Each data name description includes the following sections (if applicable):

- Synopsis* shows the declaration for the data name, including any header files that must be included in the calling routine.
- Description* describes what the data name does, what it is used for, and which functions use it. This section also contains any additional details you need to use the data name.
- Portability* describes the systems or languages with which the data name is compatible. A data name's portability falls into one of these categories:
 - AmigaDOS indicates the data name can be used only under Commodore's AmigaDOS.
 - ANSI indicates the data name is compatible with the definition in the ANSI Standard for C.
 - SAS/C indicates the name is in the SAS/C portable library. These data names are available only when using the SAS/C libraries and may or may not be equivalent to data names in libraries from other vendors.
 - UNIX indicates the data name is compatible with AT&T's UNIX System V or BSD 4.x UNIX.
- Example* contains sample code showing how to use the data name.
- See Also* lists related functions and data names.

__BackGroundIO Route I/O to the console window

Synopsis `long __BackGroundIO;`

Description The global variable `__BackGroundIO` is used by `cback.o` to indicate whether output will be sent to the console window by your background process. The default is 0, indicating that I/O will not be sent to the console window.

Setting `__BackGroundIO` to 1 tells `cback.o` to initialize `__Backstdout` to point to the console window. If `__BackGroundIO` is set to 1, you must call `Close` on `__Backstdout` before the program exits:

```
Close (__Backstdout);
```

If you do not include this statement before your program exits, the CLI window pointed to by `__Backstdout` will not close properly.

This variable is of interest only for background processes. If you do not link with `cback.o`, this variable is ignored.

Portability SAS/C

See Also `__Backstdout`

`__Backstdout` Pointer to the console window file handle

Synopsis `BPTR __Backstdout;`

Description The global variable `__BackGroundIO` is used by `cback.o` to point to the console window where I/O is to be routed. If `__BackGroundIO` is set to 1, `cback.o` initializes `__Backstdout` to point to the console window. Otherwise, `__Backstdout` is not initialized. If `__BackGroundIO` is set to 1 and `__Backstdout` is initialized, you must call `Close` for `__Backstdout` before the program exits.

```
Close (__Backstdout);
```

If you do not include this statement before your program exits, the CLI window pointed to by `__Backstdout` will not close properly.

This variable is of interest only for background processes. If you do not link with `cback.o`, this variable is ignored.

Portability SAS/C

See Also `__BackGroundIO`

— **__buffsize** Level 2 I/O buffer size

Synopsis `extern int __buffsize;`

Description This external integer is used by the level 2 I/O system to specify the size of the buffers allocated for level 2 files. This location is also used to specify the size of a buffer attached to a file with the **setbuf** or **setvbuf** function. **__buffsize** must be set to the size of the buffer before the **setbuf** or **setvbuf** function is called or before the first I/O operation; otherwise, the default buffer size is used.

The buffer is not allocated when the file is opened. Instead, the first I/O operation causes the buffer to be allocated from the local memory pool if one has not been previously specified with the **setbuf** or **setvbuf** function. This means that if **__buffsize** is changed between the **fopen** call and the first I/O operation, the size of the buffer allocated for the file will be the value of **__buffsize** at the time of the I/O operation, not the value when the file was opened.

In Version 5.10 and earlier, this variable was named **__bufsiz**.

Portability SAS/C

See Also **fopen**, **setbuf**

__ctype Character class table

Synopsis `#include <ctype.h>`

Description The external character array **__ctype** is a table indicating the attributes of all ASCII characters. It is 257 bytes long; one byte per ASCII character plus one to handle the EOF character. The individual bits within the byte are set to indicate attributes as described below.

Bit	Value	Meaning
_U	1	uppercase alphabetic character (A-Z)
_L	2	lowercase alphabetic character (a-z)
_N	4	decimal digit (0-9)
_S	8	white space
_P	16	punctuation character
_C	32	control character
_B	64	blank
_X	128	hexadecimal digit (0-9, A-F, a-f)

All indexes are offset by one so that the table also handles the end-of-file character (-1). You must index the table by the character plus one.

Normally, you will not need to access the array directly, but will instead use the `is...` character test macros (`isupper`, `isascii`, and others) to access it for you.

In Version 5.10 and earlier, this variable was named **__ctype**.

Portability SAS/C

Example `#include <ctype.h>`

```
/* Modify the ctype array to treat the character 0x80 as */
/* white space. Change 0x81 to be an uppercase character. */
/* This means isspace() will return TRUE on 0x80, and */
/* isupper(), isalpha(), isalnum(), isprint(), and isgraph() */
/* will return TRUE on 0x81. */
```

— **__ctype** Character class table
(continued)

```
void main(void)
{
    /* Make 0x80 a white space character */
    __ctype[0x80+1] |= _S;

    /* Make 0x81 an uppercase character */
    __ctype[0x81+1] |= _U;
}
```

See Also `isalnum`, `isalpha`, `isascii`, `isctrl`, `isdigit`,
`isgraph`, `islower`, `isprint`, `ispunct`, `isspace`,
`isupper`, `isxdigit`

__daytbl Array of days by name

Synopsis `extern char *__daytbl[];`

Description The external array `__daytbl` contains seven pointers to character strings representing the days of the week. Each string is three bytes long, plus a null terminator. The values are as follows:

Index	String	Meaning
0	Sun	Sunday
1	Mon	Monday
2	Tue	Tuesday
3	Wed	Wednesday
4	Thu	Thursday
5	Fri	Friday
6	Sat	Saturday

Portability SAS/C

Example

```
/*
 * Replace the day table with a French version
 * You might do this as a static DATA initialization if you
 * know you will be in French; or, you could call this
 * routine if and when the user selects French from a menu
 * item somewhere
 */
```

```
extern char *__daytbl[];
```

```
void main (void)
```

```
{
    __daytbl[0] = "Dim"; /* Dimanche (Sunday) */
    __daytbl[1] = "Lun"; /* Lundi (Monday) */
    __daytbl[2] = "Mar"; /* Mardi (Tuesday) */
    __daytbl[3] = "Mer"; /* Mercredi (Wednesday) */
    __daytbl[4] = "Jeu"; /* Jeudi (Thursday) */
}
```


— **__daytbl** Array of days by name
(continued)

```
        __daytbl[5] = "Ven",  /* Vendredi (Friday)    */  
        __daytbl[6] = "Sam"; /* Samedi (Saturday)  */  
    }
```

See Also `__montbl`, `__months`

DOSBase AmigaDOS library vector

Synopsis `extern struct DosLibrary *DOSBase;`
 `DOSBase = OpenLibrary("dos.library", revision);`

Description This external location is used by various Amiga library routines that interface with AmigaDOS system functions. It is initialized by an **OpenLibrary** call in the startup code and is expected to contain the base address of the AmigaDOS system library vector table. If you do not link with the startup module `c.o`, and if you make calls to AmigaDOS system functions or use SAS/C features that call AmigaDOS system functions, then you must first initialize this location by calling the **OpenLibrary** function.

 If you call the **OpenLibrary** function, you must close the library before your program terminates.

 For information about the *revision* parameter, see “Defining Library Bases” in Chapter 2, “Using the Commodore Libraries.”

Portability AmigaDOS

Example `/* This example demonstrates how to use OpenLibrary on */`
 `* dos.library to initialize DOSBase. It finishes by */`
 `* closing dos.library, as all programs which open it */`
 `* must do before terminating.                                */`

`#include <stdio.h>`
 `#include <stdlib.h>`

`extern struct DosLibrary *DOSBase;`

`void main(void)`
 `{`
 `int ver = 34; /* Revision number for AmigaDOS version 1.3 */`
 `DOSBase = (struct Library *) OpenLibrary("dos.library", ver);`
 `/* Make sure library opened OK */`

`if (DOSBase == NULL)`
 `exit(EXIT_FAILURE);`

`/* Insert your code using DOSBase here ... */`

`CloseLibrary ((struct Library *)DOSBase);`
 `}`

errno UNIX error number

Synopsis `#include <error.h>`

```
extern int errno;
extern int __sys_nerr;
extern char *__sys_errlist[];
```

Description The external integer **errno** is initialized to 0 at start-up time. Then, if an error is detected by one of the standard library functions, a nonzero value is placed there. The standard library never resets **errno**.

Programmers typically use this information in two ways. In some cases, it is appropriate to check **errno** after a sequence of operations and abort if any error occurred along the way. In other cases, **errno** is checked periodically, and if it is nonzero, the appropriate corrective action is taken. Then, the application program resets **errno** before beginning the next processing phase.

The **__sys_nerr** and **__sys_errlist** items are defined in a C source file named **syserr.c** and are used by the **perror** function to print messages that correspond to the code found in **errno**.

Even though error information is normally placed into **errno** by the standard library functions, application programs can also use this technique to indicate problems. However, you should be careful about adding new codes and messages just above the highest UNIX code currently defined, since new UNIX codes are occasionally added. Also, it is recommended that you add application-dependent codes by extending the header file **error.h**, which contains symbolic definitions of the code numbers. The currently defined symbols are as follows:

Symbol	Meaning
EOSERR	operating system error
EPERM	user is not owner
ENOENT	no such file or directory
ESRCH	no such process

(continued)

errno UNIX error number
(continued)

Symbol	Meaning
EINTR	interrupted system call
EIO	I/O error
ENXIO	no such device or address
E2BIG	arg list is too long
ENOEXEC	EXEC format error
EBADF	bad file number
ECHILD	no child process
EAGAIN	no more processes allowed
ENOMEM	no memory available
EACCES	access denied
EFAULT	bad address
ENOTBLK	block device required
EBUSY	resource is busy
EEXIST	file already exists
EXDEV	cross-device link
ENODEV	no such device
ENOTDIR	is not a directory
EISDIR	is a directory
EINVAL	invalid argument
ENFILE	no more files (units) allowed
EMFILE	no more files (units) allowed for this process

(continued)

errno UNIX error number
(continued)

Symbol	Meaning
ENOTTY	not a terminal
ETXTBSY	text file is busy
EFBIG	file is too large
ENOSPC	no space left
ESPIPE	seek issued to pipe
EROFS	read-only file system
EMLINK	too many links
EPIPE	broken pipe
EDOM	math function argument error
ERANGE	math function result is out of range

Portability ANSI

See Also `__FPERR`, `perror`

__fmask Default protection bits for opening a file

Synopsis `extern unsigned long __fmask;`

Description This external integer indicates the default protection for any file created by the SAS/C file I/O routines, including all ANSI I/O routines. This flag has no effect on AmigaDOS I/O routines such as **Open**. The default protection is read, write, and delete.

You can modify **__fmask**. To reset **__fmask**, use the following values (defined in the `fcntl.h` file):

Value	Meaning
<code>S_ISCRIPT</code>	script
<code>S_IPURE</code>	pure
<code>S_IARCHIVE</code>	archive
<code>S_IREAD</code>	read
<code>S_IWRITE</code>	write
<code>S_IEXECUTE</code>	execute
<code>S_IDELETE</code>	delete

Portability SAS/C

See Also `creat`, `fopen`, `open`

__fmode Default level 2 I/O mode

Synopsis `extern int __fmode;`

Description This external integer is used by the `fopen` function to determine the translation mode to use when the programmer does not specify a mode in the `fopen` call. For AmigaDOS, it is set to `O_RAW`, which specifies untranslated mode.

Portability SAS/C

See Also `fopen`

__FPERR Floating-point error code

Synopsis `#include <math.h>`

Description **__FPERR** contains a nonzero value after any low-level floating-point operation encounters an error. Low-level operations include addition, subtraction, multiplication, division, comparison, and conversion from one number representation to another (for example, floating point to double-precision floating point).

The `math.h` file contains the declaration `extern int __FPERR`, so you do not need to include this statement in your program.

The following table lists the error codes and their corresponding symbols from the `math.h` file:

Symbol	Value	Meaning
<code>__FPEUND</code>	1	underflow
<code>__FPEOVF</code>	2	overflow
<code>__FPEZDV</code>	3	divide by zero
<code>__FPENAN</code>	4	not a valid number
<code>__FPECOM</code>	5	not comparable

When the error occurs, the low-level operation passes the appropriate error code to the `__CXFERR` function, which must store the code in `__FPERR`. `__FPERR` is never reset by any low-level operation.

For compatibility with previous releases, the error code values without the leading underscore are also defined in the `math.h` file unless you define the `__STRICT_ANSI` flag before including `math.h`.

Note: This variable is only set when using the standard math libraries `scm.lib` and `scms.lib`.

Portability SAS/C

Example

```

/*
 * This example uses the division operation to produce
 * floating point errors. For example, Enter 0 for the
 * divisor and 1 for the dividend to produce
 * __FPERR = 3 (Divide by zero).
 */

```


__FPERR Floating-point error code
(continued)

```
#include <math.h>
#include <stdio.h>

void main (void)
{
    double a,b,c;

    while(feof(stdin) == 0)
    {
        printf("Enter divisor: ");
        if(scanf("%lf",&a) != 1)
            exit(0);

        printf("Enter dividend: ");
        if(scanf("%lf",&b) != 1)
            exit(0);

        __FPERR = 0;

        c = b / a;
        printf("__FPERR = %d\n",__FPERR);
        printf("%lf / %lf = %lf\n\n",b,a,c);
    }
}
```

See Also **errno**

GfxBase Graphics library vector

Synopsis `extern struct GfxBase *GfxBase;
GfxBase = OpenLibrary("graphics.library", revision);`

Description This external location is used by various Amiga library routines that interface with the ROM Kernel graphics system functions. It must be initialized by an `OpenLibrary` call before any of the graphics functions documented in the Amiga ROM Kernel manuals can be called. It is expected to contain the base address of the graphics library vector table.

You must close the library before your program terminates.

For information about the *revision* parameter, see “Defining Library Bases” in Chapter 2, “Using the Commodore Libraries.”

Portability AmigaDOS

Example `/* This example demonstrates how to use OpenLibrary on */
/* graphics.library to initialize GfxBase. It finishes */
/* by closing graphics.library, as all programs */
/* which open it must do before terminating. */

#include <stdio.h>
#include <stdlib.h>

extern struct GfxBase *GfxBase;

void main(void)
{
 /* Revision number for AmigaDOS version 1.3 */
 int ver = 34;
 GfxBase = (struct Library *)
 OpenLibrary("graphics.library", ver);
 /* Make sure library opened OK */
 if (GfxBase == NULL)
 exit(EXIT_FAILURE);

 /* Insert your code using GfxBase here ... */

 CloseLibrary ((struct Library *)GfxBase);
}`

IntuitionBase Intuition library vector

Synopsis `extern struct IntuitionBase *IntuitionBase;
IntuitionBase = OpenLibrary("intuition.library", revision);`

Description This external location is used by various Amiga library routines that interface with the Intuition system functions. It must be initialized by an **OpenLibrary** call before any of the functions documented in the Amiga Intuition manuals can be called. It is expected to contain the base address of the Intuition library vector table.

You must close the library before your program terminates.

For information about the *revision* parameter, see “Defining Library Bases” in Chapter 2, “Using the Commodore Libraries.”

Portability AmigaDOS

Example

```
/* This example demonstrates how to use OpenLibrary on */
/* intuition.library to initialize IntuitionBase. It */
/* finishes by closing intuition.library, as all */
/* programs which open it must do before terminating. */

#include <stdio.h>
#include <stdlib.h>

extern struct IntuitionBase *IntuitionBase;

void main(void)
{
    /* Revision number for AmigaDOS version 1.3 */
    int ver = 34;
    IntuitionBase = (struct Library *)
        OpenLibrary("intuition.library", ver);
    /* Make sure library opened OK */
    if (IntuitionBase == NULL)
        exit(EXIT_FAILURE);

    /* Insert your code using IntuitionBase here ... */

    CloseLibrary ((struct Library *)IntuitionBase);
}
```

MathBase FFP library vector

Synopsis `extern struct Library *MathBase;`
 `MathBase = OpenLibrary("mathffp.library", revision);`

Description This external location is used to interface with the Motorola Fast Floating Point library routines provided by Amiga. It is initialized by an `OpenLibrary` call in the `__fpinit` routine when a program compiled with the `math=ffp` option starts. It contains the base address of the FFP math library vector table. If you make direct calls to the Amiga FFP functions you must first initialize this location by calling the `OpenLibrary` function.

 You must close the library before your program terminates.

 For information about the *revision* parameter, see “Defining Library Bases” in Chapter 2, “Using the Commodore Libraries.”

Portability AmigaDOS

Example `/* This example demonstrates how to use OpenLibrary on */`
 `/* mathffp.library to initialize MathBase. It finishes */`
 `/* by closing mathffp.library, as all programs        */`
 `/* which open it must do before terminating.        */`

`#include <stdio.h>`
 `#include <stdlib.h>`

`extern struct Library *MathBase;`

`void main(void)`
 `{`
 `/* Revision number for AmigaDOS version 1.3 */`
 `int ver = 34;`

 `MathBase = (struct Library *)`
 `OpenLibrary("mathffp.library", ver);`

 `/* Make sure library opened OK */`
 `if (MathBase == NULL)`
 `exit(EXIT_FAILURE);`

 `/* Insert your code using Amiga FFP functions here ... */`

 `CloseLibrary ((struct Library *)MathBase);`
 `}`

MathTranBase FFP transcendental library vector

Synopsis `extern struct Library *MathTranBase;`
`MathTranBase = OpenLibrary("mathtrans.library", revision);`

Description This external location is used to interface with the Motorola Fast Floating Point format transcendental function library routines provided by Amiga. It is initialized by an `OpenLibrary` call when a program compiled with the `FFP` option starts. It contains the base address of the FFP transcendental math library vector table. If you make direct calls to the Amiga FFP transcendental functions you must first initialize this location by calling the `OpenLibrary` function.

You must close the library before your program terminates.

For information about the *revision* parameter, see “Defining Library Bases” in Chapter 2, “Using the Commodore Libraries.”

Portability AmigaDOS

Example

```

/* This example demonstrates how to use OpenLibrary on */
/* mathtrans.library to initialize MathTranBase.  It    */
/* finishes by closing mathtrans.library, as all        */
/* programs which open it must do before terminating. */

#include <stdio.h>
#include <stdlib.h>

extern struct Library *MathTranBase;

void main(void)
{
    /* Revision number for AmigaDOS version 1.3 */
    int ver = 34;
    MathTranBase = (struct Library *)
        OpenLibrary("mathtrans.library", ver);

    /* Make sure library opened OK */
    if (MathTranBase == NULL)
        exit(EXIT_FAILURE);
    /* Insert your code using Motorola Fast Floating Point format
       transcendental functions here ... */
    CloseLibrary ((struct Library *)MathTranBase);
}

```

MemType Type of memory desired

Synopsis `#include <exec/memory.h>`
 `extern unsigned long _MemType;`

Description The external long integer **MemType** represents the type memory to be allocated by any of the memory allocation routines. The default is **MEMF_ANY**. You can set **MemType** to any of the following mnemonics:

Mnemonic	Value
MEMF_ANY	0L (any type of memory will do)
MEMF_PUBLIC	1L<<0
MEMF_CHIP	1L<<1
MEMF_FAST	1L<<2
MEMF_LOCAL	1L<<8
MEMF_24BITDMA	1L<<9 (DMAable memory within 24 bits of address)

These mnemonics are defined in the file **exec/memory.h**.

Note: It is safe to change **MemType** at any point in a program.

Portability AmigaDOS

Example `#include <exec/memory.h>`

```

extern long _MemType;

void main (void)
{
    void *chipmem;
    void *fastmem;
    void *anymem;
    long oldtype;

    oldtype = _MemType;

```

__MemType Type of memory desired
(continued)

```

/* allocate 50 bytes of any type of memory */
__MemType = MEMF_ANY;
anymem = malloc(50);

/* allocate 50 bytes of fast memory */
__MemType = MEMF_FAST;
fastmem = malloc(50);

/* allocate 50 bytes of chip memory */
__MemType = MEMF_CHIP;
chipmem = malloc(50);

__MemType = oldtype;
}

```

See Also `malloc`, `AllocMem` in the autodocs from Commodore

— **__montbl** Array of months by name

Synopsis `extern char *__montbl[];`

Description The external array `__montbl` contains twelve pointers to character strings representing the months of the year. Each string is three bytes long, plus a null terminator. The values are as follows:

Index	String	Meaning
0	Jan	January
1	Feb	February
2	Mar	March
3	Apr	April
4	May	May
5	Jun	June
6	Jul	July
7	Aug	August
8	Sep	September
9	Oct	October
10	Nov	November
11	Dec	December

Portability SAS/C

Example

```

/*
 * Replace the month table with a German version
 * You might do this as a static DATA initialization if you
 * know you will be in German; or, you could change this into
 * a function that would be called if and when the user
 * selects German from a menu item somewhere
 */

extern char *__montbl[];
```


— **__montbl** Array of months by name
(continued)

```
void main (void)
{
    __montbl[ 0] = "Jan"; /* Januar    (January)  */
    __montbl[ 1] = "Feb"; /* Februar  (February) */
    __montbl[ 2] = "Mar"; /* Marz     (March)     */
    __montbl[ 3] = "Avr"; /* Avril    (April)     */
    __montbl[ 4] = "Mai"; /* Mai      (May)       */
    __montbl[ 5] = "Jun"; /* Juni     (June)      */
    __montbl[ 6] = "Jul"; /* Juli     (July)      */
    __montbl[ 7] = "Aug"; /* August   (August)    */
    __montbl[ 8] = "Sep"; /* September (September) */
    __montbl[ 9] = "Okt"; /* Oktober  (October)   */
    __montbl[10] = "Nov"; /* November (November)  */
    __montbl[11] = "Dez"; /* Dezember (December)  */
}
```

See Also `__daytbl`, `__months`

__months Array of month lengths

Synopsis `extern char __months[];`

Description The external array `__months` contains twelve characters, each representing the number of days in a specific month (in a non-leap year.) The values are as follows:

Index	Value	Month
0	31	January
1	28	February
2	31	March
3	30	April
4	31	May
5	30	June
6	31	July
7	31	August
8	30	September
9	31	October
10	30	November
11	31	December

Your programs must account for leap years when dealing with February.

Portability SAS/C

Example

```

/*
 * This is a sample program demonstrating the use of __months[].
 * Count the number of days since a given month and day. Assume
 * both dates are within the same calendar year. Assume month and
 * date are BEFORE thismonth and thisdate.
 */

```

— **__months** Array of month lengths
(continued)

```
#include <stdio.h>

extern char __months[];

int countdays(int month, int date, int thismonth, int thisdate)
{
    int days;
    int curmonth;

    if (thismonth == month)
        days = thisdate - date;

    else
    {
        days = __months[month] - date;

        for(curmonth = month + 1; thismonth > curmonth; curmonth++)
        {
            days += __months[curmonth];
        }

        days += thisdate;
    }
    return(days);
}

void main (void)
{
    int x;

    /* Call countdays, print the result */
    x = countdays(7,12,10,5);
    printf ("x = %d \n",x);
}
```

See Also `__daytbl`, `__monttbl`

— **__msflag** MS-DOS file pattern flag

Synopsis `extern int __msflag;`

Description This external integer is used by the filename pattern matching functions to specify AmigaDOS or MS-DOS wildcard characters. If **__msflag** is nonzero, MS-DOS filename patterns are used. By default, AmigaDOS filename patterns are used.

Portability SAS/C

See Also `dfind`, `getfnl`

`__MSTEP` Memory pool increment size

Synopsis `extern long __MSTEP;`

Description This external integer is used by the memory allocation functions. It specifies the minimum amount of memory that will be allocated from the system for the local memory pool. The default value is 16,384 bytes.

While additional memory is added to the local pool, it will not be contiguous with the memory already in the pool. If the additional amount is small, it can lead to severe fragmentation of the local pool. The memory allocation functions attempt to avoid this by rounding up the amount needed to the next multiple of the figure in `__MSTEP`.

This technique works well for small allocation requests. However, if your application requires mostly large blocks of memory, `__MSTEP` should be set to a small nonzero figure to allow for a more efficient allocation.

Portability SAS/C

See Also `free`, `malloc`

__nufbs Count of level 1 file handle slots

Synopsis `#include <ios1.h>`

`extern int __nufbs;`

Description The external integer **__nufbs** indicates how many level 1 file handles exist in the **__ufbs** linked list.

In Version 5.10 and earlier, this data name was called **_nufbs**.

Portability SAS/C

See Also **__ufbs**

_OSERR DOS error information

Synopsis `#include <dos.h>`

```
extern int _OSERR;
    /* Number of AmigaDOS error codes */
extern int __os_nerr;
    /* AmigaDOS error messages */
extern struct DOS_ERRS __os_errlist[];
```

Description The external integer `_OSERR` contains error information returned by AmigaDOS after a system call has failed. In general, the SAS/C library resets `_OSERR` at the beginning of any function that makes AmigaDOS system calls. Then, if a system call fails during that function, the system error code is saved in `_OSERR`.

The AmigaDOS error number is mapped into an equivalent UNIX error number, which is placed in `errno`. If there is no appropriate UNIX number, `errno` will contain `-1`, defined symbolically as `EOSERR`.

The `__os_nerr` and `__os_errlist` items are defined in a C source file named `oserr.c` and are used by the `poserr` function to print messages that correspond to the code found in `_OSERR`.

The following list of system error codes applies to AmigaDOS 2.0 and is contained in the file `_oserr.c`. `#define` values for these codes are in the file `dos/dos.h`.

Code	Meaning
103	not enough memory is available
105	process table is full
114	bad template
115	bad number
116	required argument is missing
117	value after keyword is missing

(continued)

_OSERR DOS error information
(continued)

Code	Meaning
118	wrong number of arguments
119	unmatched quotes
120	argument line is invalid or too long
121	file is not executable
122	invalid resident library
202	object is in use
203	object already exists
204	directory not found
205	object not found
206	invalid window description
207	object is too large
209	packet request type is unknown
210	object name is invalid
211	invalid object lock
212	object is not of required type
213	disk is not validated
214	disk is write-protected
215	rename across devices attempted
216	directory is not empty
217	too many levels

(continued)

_OSERR DOS error information
(continued)

Code	Meaning
218	device (or volume) is not mounted
219	seek failure
220	comment is too long
221	disk is full
222	object is protected from deletion
223	file is write-protected
224	file is read-protected
225	not a valid DOS disk
226	no disk is in drive
232	no more entries are in directory

The following error codes are defined only under AmigaDOS 2.0 and are documented in the `dos/dos.h` file.

Code	Meaning
233	object is soft link
234	object is linked
235	bad loadfile hunk
236	function not implemented
240	record is not locked
241	record lock collision
242	record lock timeout
243	record unlock error

The following errors occur only when calling the `MatchFirst` function or `MatchNext` function or both functions of `dos.library`

_OSERR DOS error information
(continued)

under AmigaDOS 2.0. These errors are documented in the file **dos/dos.as1**.

Code	Meaning
303	buffer overflow
304	break character received
305	not executable

Portability AmigaDOS

See Also AmigaDOS Technical Reference Manual, **errno**, **poserr**

— **__priority** Default priority for a background program

Synopsis `long __priority=-5;`

Description The global variable `__priority` is used by the `cback.o` startup code to indicate the priority at which to start the background program. A value of 0 is the normal priority while larger values run at higher priority. This number is limited to the range `-128` to `127`, inclusive.

You must initialize the variable at compile time. Setting this variable after the program has started does not affect the program priority.

If you do not declare a `__priority` variable, `cback.o` defaults to a priority of 0 (based on a default value drawn from the library).

If you do not link with `cback.o`, this variable is ignored.

Portability AmigaDOS

Example

```

/* A priority of 50 will ensure that your background program
 * runs. A priority this high will even prevent input
 * handlers from running, so the machine will be effectively
 * crashed by doing this (until your program calls
 * something that causes it to pause).
 *
 **** REMEMBER: You must link with cback.o for this variable
 ****           to have any effect.
 */

long __priority=50;

void main (void)
{
    /* Insert your program here */
}
```

See Also `__procname`, `__stack`

__procname Process name for a spawned program

Synopsis `char *__procname="MyBackgroundProcess";`

Description The global variable `__procname` is the name that `cback.o` uses when spawning a background process. You may choose any name but should stick to something that is meaningful for your program.

You must initialize this variable at compile time. Setting this variable after the program has started has no effect on the program name. Also, if the program is started from the workbench, the `__procname` parameter is ignored. If you do not link with `cback.o`, this variable is ignored.

Portability AmigaDOS

Example

```
/*
 * Set the name of the background process.
 *
 * ***** REMEMBER: You must link with cback.o for this variable
 * ***** to have any effect.
 */

char *__procname="SpecialProcess";

void main (void)
{
    /* Insert your program here */
}
```

See Also `__priority`, `__stack`

— **__sigfunc** Signal function table

Synopsis `#include <signal.h>`

```
extern void (*__sigfunc[NSIG])(int);
```

Description **__sigfunc** is the array of all signal handles. When the **raise** function is called, it looks in the table to see what is installed at the corresponding slot in the **__sigfunc** array. Each entry is either a pointer to the function to be called when the event occurs or one of two special values:

SIG_IGN ignore the condition.

SIG_DFL take the default action.

There are **NSIG** elements in the array corresponding to the predefined signal values from the file **signal.h**.

Symbol	Value	Meaning
SIGABRT	1	abnormal termination, abort
SIGFPE	2	floating-point exception
SIGILL	3	illegal instruction
SIGINT	4	interrupt from AmigaDOS, Control-C or Control-D
SIGSEGV	5	segmentation violation (not generated on the Amiga)
SIGTERM	6	termination request

Signals 0, 7, and 8 are reserved for future use (and for compatibility with POSIX implementations).

You should not set elements of this array directly. Use the **signal** function instead.

Portability UNIX

See Also **raise**, **signal**

__stack Minimum program stack size

Synopsis `long __stack = 8192;`

Description The global variable `__stack` is used by the startup code to determine the minimum stack space necessary to run a program. You must initialize the variable at compile time. Setting this variable after the program has started does not affect the stack size.

 If the startup code determines that there is not enough stack space to satisfy the request, it allocates a temporary stack and runs the program on that stack.

Portability SAS/C

Example

```

/*
 * Make sure we have a LOT of stack space to run.
 */

long __stack = 25000L;

void main (void)
{
    /* Insert your program here */
}
```

See Also `__priority`, `__procname`

__sys_errlist Errno text strings

Synopsis `#include <errno.h>`

```
extern char *__sys_errlist[];
```

Description The external array of strings **__sys_errlist** provides the text message that corresponds to a given **errno** value. The value of **errno** is used as a direct index to this table. You should avoid accessing this table directly to print errors. Instead, use the **perror** or **strerror** functions.

You may want to substitute your own table for internationalization of your program. The external integer **__sys_nerr** indicates the number of entries in **__sys_errlist**.

If you change **__sys_errlist** dynamically in your program, as shown in the example below, the changes are only in effect for that program while it is running. To change the messages permanently for all programs, modify **syserr.c** in the source directory, compile it, and use the OML to replace it in all nonmath libraries that you plan to use. As an alternative, you can compile the **syserr.c** file and link it with all programs in which you want modified messages.

In Version 5.10 and earlier, this variable was named **sys_errlist**.

Portability SAS/C

Example

```
/* Print error message number 1 with perror, change the */
/* message to all capitals, and then print the          */
/* message again.                                       */

#include <errno.h>

void main (void)
{
    const char *x = "Test Message";
    errno = 1;
    perror(x);
    __sys_errlist[errno] = "USER IS NOT OWNER";
    perror(x);
}
```

See Also **perror**, **strerror**, **__sys_nerr**

__sys_nerr Number of entries in __sys_errlist

Synopsis `#include <errno.h>`

`extern int __sys_nerr;`

Description The external integer `__sys_nerr` indicates the number of entries in the `__sys_errlist` array.

In Version 5.10 and earlier, this variable was named `sys_nerr`.

Portability SAS/C

Example

```
/*
 * Print all possible errors from the __sys_errlist array
 */

#include <errno.h>

void main (void)
{
    int i;

    for ( i=0; i <= __sys_nerr; i++)
    {
        printf("Message %d = %s \n",i,strerror(i));
    }
}
```

See Also `perror`, `strerror`, `__sys_errlist`

__TZ Default time zone name

Synopsis `#include <time.h>`

```
extern char *__TZ;
```

Description This variable is the default time zone that is used if there is no environment variable **TZ** set on the Amiga. If you set it, you must make sure that the memory for it is not freed until you reset it. If **__TZ** is not set to any value, **CST6** is used instead.

The value of the variable **__TZ** is used to set other time zone variables, including **__tzstn**, **__tzdtn**, and **__daylight**. The first three characters are taken as the name. The remaining characters are taken as the ASCII representation of the number of hours offset from Greenwich Mean Time (GMT) for the time zone.

Portability SAS/C

Example

```
/*
 * Print out the default time zone and then set it to be EST.
 */

#include <time.h>
#include <stdio.h>

void main(void)
{
    printf("Default Time Zone = %s\n", __TZ);
    __TZ = "EST5";
}
```

See Also `__tzset`

— **__ufbs** Array of open level 1 file handles

Synopsis `#include <ios1.h>`

```
extern struct UFB *__ufbs;
```

Description `__ufbs` is used to track all open level 1 file handles. `__ufbs` is the pointer to the beginning of the linked list containing the file handles. If `__ufbs` is equal to NULL, then there are no open level 1 files.

The `open` or `creat` function returns the entry number of the file handle in the linked list. For example, if the `open` function returns 5, the file handle is the fifth entry of the linked list.

It is not recommended that you access this linked list directly. The `chkufb` function is provided to translate an index into the appropriate UFB structure. The UFB structure actually consists of four members:

```
struct UFB
{
    struct UFB *ufbnxt;    /* next ufb */
    int ufbflg;           /* flags */
    long ufbfh;           /* file handle */
    char *ufbfname;       /* file name */
};
```

The `ufbflg` field consists of bits as defined here:

`UFB_RA` indicates that the file may be read from.

`UFB_WA` indicates that the file may be written to.

`UFB_NC` indicates that the file handle should not be closed on exit.

`UFB_APP` indicates that the file may be appended to on write.

`UFB_XLAT` indicates file translation mode.

`UFB_TEMP` indicates the file is a temporary file.

In Version 5.10 and earlier, the linked list `__ufbs` was an array named `_ufbs`.

Portability SAS/C

See Also `chkufb`, `__nufbs`, `open`

7 Library Reference

This chapter describes the library functions provided with the SAS/C software.

The functions are described in alphabetical order. Each function description includes the following sections (if applicable):

Synopsis shows the declaration for the function, including any header files that must be included in the calling routine.

Note: The keyword `const` is used in the declarations of many of the variables referenced by the library functions. This keyword defines variables as read only, and its use in the libraries ensures that the library functions do not change the values of these variables. Do not use the keyword `const` in variable declarations unless you want to prevent your program from changing the value of that variable. The compiler issues an error message if you try to modify the contents of a variable declared with the keyword `const`.

Description describes what the function does and contains all the information you need to use the function. This section identifies all functions that are not available if you define the `__STRICT_ANSI` preprocessor symbol. For more information, see the following section, “The `__STRICT_ANSI` Symbol.”

Portability describes the systems or languages with which the function is compatible. A library function’s portability falls into one of these categories:

AmigaDOS indicates the function can be used only on machines running AmigaDOS.

ANSI indicates the function is compatible with the definition in the ANSI Standard for C.

SAS/C indicates the function is in the SAS/C Development System library. SAS/C functions are available only when using the SAS/C libraries and may or may not be equivalent to functions in libraries from other vendors.

OLD	indicates the function was defined in earlier versions of the SAS/C Compiler and can still be used, although the newer ANSI and AmigaDOS functions are now preferred.
UNIX	indicates the function is compatible with AT&T's UNIX System V or BSD 4.x UNIX.
XENIX	indicates the function is compatible with Microsoft's XENIX.
<i>Returns</i>	describes any values that the function returns. If the function does not return anything, this section is not included for that function.
<i>Example</i>	contains sample code showing how to use the function. If the function is OLD, an example is not included.
<i>See Also</i>	refers you to the descriptions of data names and other functions that contain related information.

The **_STRICT_ANSI** Symbol

The ANSI Standard for C requires that certain compiler header files must be present and that these compiler header files and the libraries define certain symbols, macros, and functions. The SAS/C Development System headers and libraries define all of the required ANSI symbols. They also define many other symbols that are not required by the ANSI Standard.

To be completely ANSI-compliant, an implementation must not define any non-ANSI symbols in an ANSI header file unless the symbols begin with an underscore followed by a capital letter or an underscore followed by another underscore.

If you define the preprocessor symbol **_STRICT_ANSI**, the SAS/C Development System headers do not define any symbols, macros, or functions that violate the ANSI Standard naming conventions. You can use a **#define** statement to define the **_STRICT_ANSI** symbol, or you can define it on the command line. If you define the **_STRICT_ANSI** symbol, the compiler does not include the prototypes or **#define** statements for some library functions and macros. The descriptions in this chapter identify which functions and macros are affected by the **_STRICT_ANSI** preprocessor symbol.

Function Categories

The following lists contain the names, descriptions, and portability classifications of all of the functions described in this manual. These

functions are grouped by category such as string functions or I/O functions. Within each category, functions that perform similar tasks are listed together. For example, `stpchrn` and `strchr` both perform the same function, however, `stpchrn` is an OLD function and `strchr` is an ANSI function. Some functions may be listed twice. For example, the `ctime` function is listed under “Conversion Functions” and under “Time Functions.”

Character-Type Macros and Functions

Function	Portability	Description
<code>isalnum</code>	ANSI	check if a character is alphanumeric
<code>isalpha</code>	ANSI	check if a character is alphabetic
<code>isascii</code>	SAS/C	check if a character is an ASCII character
<code>isctrl</code>	ANSI	check if a character is a control character
<code>iscsym</code>	SAS/C	check if a character is a C symbol character
<code>iscsymf</code>	SAS/C	check if a character is a C symbol lead character
<code>isdigit</code>	ANSI	check if a character is a decimal digit (0-9)
<code>isgraph</code>	ANSI	check if a character is a graphic character
<code>islower</code>	ANSI	check if a character is lowercase
<code>isprint</code>	ANSI	check if a character is printable
<code>ispunct</code>	ANSI	check if a character is a punctuation character
<code>isspace</code>	ANSI	check if a character is a space
<code>isupper</code>	ANSI	check if a character is uppercase
<code>isxdigit</code>	ANSI	check if a character is a hex character (0-9, A-F, a-f)

(continued)

Function	Portability	Description
toascii	SAS/C	convert a character to an ASCII character
tolower	ANSI	convert a character to lowercase
toupper	ANSI	convert a character to uppercase

String Functions See also “Memory Manipulation Functions,” later in this section.

Function	Portability	Description
astcsma	AmigaDOS	AmigaDOS string pattern match (anchored)
stcpm	SAS/C	unanchored pattern match
stcpma	SAS/C	anchored pattern match
stcsma	AmigaDOS	UNIX string pattern match (anchored)
stccpy	UNIX	copy one string to another
stpcpy	SAS/C	copy one string to another
strcpy	ANSI	copy one string to another
strncpy	ANSI	copy a string, length limited
stcis	SAS/C	count the number of string characters in the set
stcisin	SAS/C	count the number of string characters not in the set
strcspn	ANSI	count the number of string characters not in the set
stclen	OLD	measure the length of a string
strlen	ANSI	measure the length of a string
strspn	ANSI	count the number of string characters in the set
stpbrk	OLD	find a break character in a string

(continued)

Function	Portability	Description
stpchr	OLD	find a character in a string
strchr	ANSI	find a character in a string
stpchrn	OLD	find a character not in a string
strrchr	ANSI	find a character not in a string
strpbrk	ANSI	find a break character in a string
strcat	ANSI	concatenate strings
strncat	ANSI	concatenate strings, length limited
strcmp	ANSI	compare strings, case sensitive
strcmpi	OLD	compare strings, case insensitive
stricmp	SAS/C	compare strings, case insensitive
strncmp	ANSI	compare strings, length limited
strnicmp	SAS/C	compare strings, case insensitive, length limited
strnset	XENIX	set a string to a value, length limited
strset	XENIX	set a string to a value
strtok	ANSI	get a token
stcarg	SAS/C	get an argument
stpsym	SAS/C	get the next symbol from a string
stptok	SAS/C	get the next token from a string
stpblk	SAS/C	skip blanks
strbpl	SAS/C	build a string-pointer list
strdup	XENIX	duplicate a string
strins	SAS/C	insert a string
strrev	XENIX	reverse a character string
strmid	SAS/C	return a substring from a string
strstr	ANSI	find a substring inside of a string

Conversion Functions	Function	Portability	Description
	<code>atof</code>	ANSI	convert an ASCII string to a floating-point number
	<code>atoi</code>	ANSI	convert an ASCII string to an integer
	<code>atol</code>	ANSI	convert an ASCII string to a long integer
	<code>stcd_i</code>	SAS/C	convert a decimal string to an integer
	<code>stcd_l</code>	SAS/C	convert a decimal string to a long integer
	<code>ecvt</code>	UNIX	convert a double-precision floating-point number to a string
	<code>fcvt</code>	UNIX	convert a floating-point number to a string
	<code>gcvt</code>	UNIX	convert a floating-point number to a string
	<code>stch_i</code>	SAS/C	convert a hexadecimal string to an integer
	<code>stch_l</code>	SAS/C	convert a hexadecimal string to a long integer
	<code>stci_d</code>	SAS/C	convert an integer to a decimal string
	<code>stci_h</code>	SAS/C	convert an integer to a hexadecimal string
	<code>stci_o</code>	SAS/C	convert an integer to an octal string
	<code>stcl_d</code>	SAS/C	convert a long integer to a decimal string
	<code>stcl_h</code>	SAS/C	convert a long integer to a hexadecimal string
	<code>stcl_o</code>	SAS/C	convert a long integer to an octal string
	<code>stco_i</code>	SAS/C	convert an octal string to an integer
	<code>stco_l</code>	SAS/C	convert an octal string to a long integer
	<code>stcu_d</code>	SAS/C	convert an unsigned integer to a decimal string
	<code>stcul_d</code>	SAS/C	convert an unsigned long integer to a decimal string

(continued)

Function	Portability	Description
<code>strlwr</code>	XENIX	convert a string to lowercase
<code>strtod</code>	ANSI	convert a string to a double-precision floating-point number
<code>strtol</code>	ANSI	convert a string to a long integer
<code>strtoul</code>	ANSI	convert a string to an unsigned long integer
<code>toascii</code>	SAS/C	convert a character to an ASCII string
<code>tolower</code>	ANSI	convert a character to lowercase
<code>toupper</code>	ANSI	convert a character to uppercase
<code>mbstowcs</code>	ANSI	convert a multibyte string to a wide-character string
<code>wcstombs</code>	ANSI	convert a wide-character string to a multibyte string
<code>stpdate</code>	SAS/C	convert a date array to a string
<code>stptime</code>	SAS/C	convert a time array to a string
<code>mktime</code>	ANSI	convert a broken down time to a <code>time_t</code> value
<code>__timecv</code>	AmigaDOS	convert a <code>time_t</code> value to an AmigaDOS DateStamp
<code>ctime</code>	ANSI	convert a <code>time_t</code> value to an ASCII string
<code>utpack</code>	SAS/C	pack UNIX time
<code>utunpk</code>	SAS/C	unpack UNIX time

Mathematical Macros and Functions

Function	Portability	Description
<code>abs</code>	ANSI	absolute value
<code>acos</code>	ANSI	arccosine function
<code>asin</code>	ANSI	arcsine function
<code>atan</code>	ANSI	arctangent function
<code>atan2</code>	ANSI	arctangent of x/y

(continued)

Function	Portability	Description
ceil	ANSI	get floating-point limits
cos	ANSI	cosine function
cosh	ANSI	hyperbolic cosine function
cot	UNIX	cotangent function
div	ANSI	compute the quotient and the remainder
exp	ANSI	exponential function
fabs	ANSI	floating-point or double-precision floating-point absolute value
floor	ANSI	get the floor of a real number
fmod	ANSI	floating-point modulus operations
frexp	ANSI	split floating-point value
iabs	SAS/C	integer absolute value
labs	ANSI	long integer absolute value
ldexp	ANSI	combine floating-point value
ldiv	ANSI	return the long integer quotient and the remainder
log	ANSI	natural logarithm function
log10	ANSI	base 10 logarithm function
max	UNIX	compute the maximum of two values
min	UNIX	compute the minimum of two values
modf	ANSI	split a floating-point value
pow	ANSI	raise a number to a power
pow2	SAS/C	raise 2 to a power
sin	ANSI	sine function
sinh	ANSI	hyperbolic sine function
sqrt	ANSI	square root function
tan	ANSI	tangent function
tanh	ANSI	hyperbolic tangent function

**Varying-Length
Argument List
Macros**

Function	Portability	Description
<code>va_arg</code>	ANSI	get an argument from a varying-length argument list
<code>va_end</code>	ANSI	end varying-length argument list processing
<code>va_start</code>	ANSI	begin varying-length argument list processing

**General Utility
Macros and
Functions**

Function	Portability	Description
<code>__emit</code>	SAS/C	emit 68000 instruction word
<code>getreg</code>	SAS/C	get 68000-specific registers
<code>putreg</code>	SAS/C	set up 68000-specific registers
<code>geta4</code>	OLD	establish addressability to the global data area
<code>isatty</code>	UNIX	test a file descriptor for a terminal device
<code>ovlyMgr</code>	AmigaDOS	overlay manager call point
<code>offsetof</code>	ANSI	get the byte offset of a structure member

Sorting Functions

Function	Portability	Description
<code>bsearch</code>	ANSI	perform a binary search
<code>dqsort</code>	SAS/C	sort an array of double-precision floating-point numbers
<code>fqsort</code>	SAS/C	sort an array of floating-point numbers
<code>lqsort</code>	SAS/C	sort an array of long integers
<code>qsort</code>	ANSI	sort a data array
<code>sqsort</code>	SAS/C	sort an array of short integers
<code>strsrt</code>	SAS/C	sort a string-pointer list
<code>tqsort</code>	SAS/C	sort an array of text pointers

**Random-Number
Generation
Functions**

Function	Portability	Description
drand48	UNIX	generate a random double-precision floating-point number (internal seed)
erand48	UNIX	generate a random double-precision floating-point number (external seed)
jrand48	UNIX	generate a random long integer (external seed)
lcong48	UNIX	set linear congruence parameters
lrand48	UNIX	generate a random positive long integer (internal seed)
mrand48	UNIX	generate a random long integer (internal seed)
nrand48	UNIX	generate a random positive long integer (external seed)
rand	ANSI	generate a random number
seed48	UNIX	set all 48 bits of an internal seed
srand	ANSI	set a seed for the rand function
srand48	UNIX	set high 32 bits of an internal seed

**Program Control
Functions**

Function	Portability	Description
abort	ANSI	abort the current process
atexit	ANSI	set an exit trap
chkabort	AmigaDOS	check for a break character
Chk_Abort	AmigaDOS	check for a break character
exit	ANSI	terminate program execution
__exit	SAS/C	terminate program execution with no clean-up
onexit	OLD	set an exit trap
XCEXIT	SAS/C	terminate the program
forkl	SAS/C	create a child process with an argument list

(continued)

Function	Portability	Description
forkv	SAS/C	create a child process with an argument vector
longjmp	ANSI	perform a long jump
setjmp	ANSI	set long jump parameters
onbreak	AmigaDOS	plant a break trap
wait	SAS/C	wait for a single child process to complete
waitm	SAS/C	wait for multiple child processes to complete
raise	ANSI	generate a signal
signal	ANSI	establish event traps

Memory Allocation Functions

Function	Portability	Description
bldmem	OLD	build a memory pool of a specified size
rstmem	OLD	reset a memory pool
sizmem	SAS/C	get the memory pool size
chkml	SAS/C	check for the largest memory block
getmem	OLD	get a memory block (short)
getml	OLD	get a memory block (long)
halloc	SAS/C	allocate a huge memory block
lsbrk	OLD	allocate memory
sbrk	OLD	allocate memory (unsigned)
calloc	ANSI	allocate and clear memory
malloc	ANSI	allocate memory
realloc	ANSI	reallocate memory
free	ANSI	free memory
_MemCleanup	AmigaDOS	deallocate all allocated memory
rbrk	OLD	release memory
rlsmem	OLD	release a memory block
rlsml	OLD	release a memory block

Memory Manipulation Functions

Function	Portability	Description
<code>memchr</code>	ANSI	find a character in a memory block
<code>memcmp</code>	ANSI	compare two memory blocks
<code>memcpy</code>	SAS/C	copy a memory block
<code>memcpy</code>	ANSI	copy a memory block
<code>memmove</code>	ANSI	copy bytes in memory
<code>memset</code>	ANSI	set a memory block to a value
<code>movmem</code>	UNIX	move a memory block
<code>repmem</code>	SAS/C	replicate values through a block
<code>setmem</code>	UNIX	set a memory block to a value
<code>swmem</code>	UNIX	swap two memory blocks

Diagnostic Control Functions

Function	Portability	Description
<code>assert</code>	ANSI	assert program validity
<code>except</code>	SAS/C	call the math error handler
<code>__matherr</code>	UNIX	math error handler
<code>perror</code>	ANSI	print an error message
<code>poserr</code>	AmigaDOS	print an AmigaDOS error message
<code>strerror</code>	ANSI	print the text for a given error number

Time Functions

Function	Portability	Description
<code>asctime</code>	ANSI	generate an ASCII time string
<code>clock</code>	ANSI	measure program processor time
<code>datecmp</code>	AmigaDOS	compare two AmigaDOS dates
<code>difftime</code>	ANSI	compute the difference between two <code>time_t</code> values
<code>gmtime</code>	ANSI	unpack Greenwich Mean Time (GMT)
<code>localtime</code>	ANSI	unpack local time
<code>mktime</code>	ANSI	convert a broken down time to a <code>time_t</code> value

(continued)

Function	Portability	Description
strftime	ANSI	format a time string
time	ANSI	get the system time in seconds
ctime	ANSI	convert a time_t value to an ASCII string
__timecvt	AmigaDOS	convert a time_t value to an AmigaDOS DateStamp
timer	AmigaDOS	get the system clock with microseconds
__tzset	XENIX	set the time zone variables
utpack	SAS/C	pack UNIX time
utunpk	SAS/C	unpack UNIX time

I/O Functions **Standard I/O Functions**

Do not use these functions with the **__tinymain** function.

Function	Portability	Description
getch	SAS/C	get a character from stdin immediately
getchar	ANSI	get a character from stdin
fgetchar	XENIX	get a character from stdin
fputchar	XENIX	put a character to stdout
putchar	ANSI	put a character to stdout
gets	ANSI	get a string from stdin
puts	ANSI	put a string to stdout
printf	ANSI	formatted print to stdout
scanf	ANSI	formatted input conversions
vprintf	ANSI	formatted write of a varying-length argument list to stdout

Character I/O Functions

Function	Portability	Description
<code>fgetc</code>	ANSI	get a character from a file
<code>getc</code>	ANSI	get a character from a file
<code>ungetc</code>	ANSI	push an input character back
<code>getch</code>	SAS/C	get a character from <code>stdin</code> immediately
<code>getchar</code>	ANSI	get a character from <code>stdin</code>
<code>fgetchar</code>	XENIX	get a character from <code>stdin</code>
<code>fputc</code>	ANSI	put a character to a level 2 file
<code>fputchar</code>	XENIX	put a character to <code>stdout</code>
<code>putc</code>	ANSI	put a character to a level 2 file
<code>putchar</code>	ANSI	put a character to <code>stdout</code>

String I/O Functions

Function	Portability	Description
<code>fgets</code>	ANSI	get a string from a level 2 file
<code>gets</code>	ANSI	get a string from <code>stdin</code>
<code>fputs</code>	ANSI	put a string to a level 2 file
<code>puts</code>	ANSI	put a string to <code>stdout</code>

Block I/O Functions

Function	Portability	Description
<code>_dread</code>	OLD	read an AmigaDOS file
<code>_dwrite</code>	OLD	write to an AmigaDOS file
<code>fread</code>	ANSI	read and write blocks
<code>fwrite</code>	ANSI	write blocks to a level 2 file
<code>read</code>	UNIX	read from a level 1 file
<code>write</code>	UNIX	write to level 1 file

Formatted I/O Functions

Function	Portability	Description
fprintf	ANSI	formatted print
printf	ANSI	formatted print to stdout
sprintf	ANSI	formatted print to a string
fscanf	ANSI	formatted input conversions
scanf	ANSI	formatted input conversions
sscanf	ANSI	formatted input conversions
vfprintf	ANSI	formatted write of a varying-length argument list to a file
vprintf	ANSI	formatted write of a varying-length argument list to stdout
vsprintf	ANSI	formatted write of a varying-length argument list to a string

Error Handling Functions

Function	Portability	Description
feof	ANSI	check for a level 2 end-of-file
ferror	ANSI	check for a level 2 error
clearerr	ANSI	clear a level 2 I/O error flag
clrerr	UNIX	clear a level 2 I/O error flag

File Control Functions

Function	Portability	Description
close	UNIX	close a level 1 file
_dclose	OLD	close an AmigaDOS file
fclose	ANSI	close a level 2 file
fcloseall	XENIX	close all level 2 files
tmpnam	ANSI	create a temporary filename
creat	UNIX	create a level 1 file
_dcreat	OLD	create an AmigaDOS file

(continued)

Function	Portability	Description
_dcreatx	OLD	create a new AmigaDOS file
fdopen	UNIX	attach a level 1 file to level 2
fileno	UNIX	get the file number for a level 2 file
fmode	SAS/C	change the mode of a level 2 file
iomode	SAS/C	change the mode of a level 1 file
open	UNIX	open a level 1 file
_dopen	OLD	open an AmigaDOS file
fopen	ANSI	open a level 2 file
tmpfile	ANSI	open a temporary file stream
freopen	ANSI	reopen a level 2 file
fflush	ANSI	flush a level 2 output buffer
flushall	XENIX	flush all level 2 output buffers
setbuf	ANSI	set the buffer mode for a level 2 file
setnbf	UNIX	turn off I/O buffering for a level 2 file
setvbuf	ANSI	set the variable buffer for a level 2 file

File Positioning Functions

Function	Portability	Description
_dseek	OLD	reposition an AmigaDOS file
fseek	ANSI	set a level 2 file position
lseek	UNIX	set a level 1 file position
fgetpos	ANSI	get the current file position
fsetpos	ANSI	reposition a file
ftell	ANSI	get a level 2 file position
tell	UNIX	get a level 1 file position
rewind	ANSI	reset a level 2 file position to its first byte

File Management Functions	Function	Portability	Description
	access	UNIX	check file accessibility
	chkufb	SAS/C	check a level 1 file handle
	fileno	UNIX	get the file number for a level 2 file
	chmod	UNIX	change a file's protection mode
	fmode	SAS/C	change the mode of a level 2 file
	iomode	SAS/C	change the mode of a level 1 file
	fstat	UNIX	get file status
	getfa	AmigaDOS	get the file attribute
	getft	AmigaDOS	get the file time
	stat	UNIX	get information on a file
	stcgfe	SAS/C	get the filename extension
	stcgfn	SAS/C	get a filename from a path
	stcgfp	SAS/C	get a file path
	strmfe	SAS/C	make a filename with an extension
	strmfn	SAS/C	make a filename from components
	strmfp	SAS/C	make a filename from path or node
	strsfn	SAS/C	split the filename
	unlink	UNIX	remove a file
	remove	ANSI	remove a file
	rename	ANSI	rename a file
	setbuf	ANSI	set the buffer mode for a level 2 file
	setnbf	UNIX	turn off I/O buffering for a level 2 file
	setvbuf	ANSI	set the buffer mode for a level 2 file

System Interface Functions

Function	Portability	Description
argopt	SAS/C	get options from an argument list
chgclk	AmigaDOS	change the system clock
getclk	AmigaDOS	get or change the system clock
getasn	AmigaDOS	get assigned device
getdfs	AmigaDOS	get disk free space
getenv	ANSI	get environment variable
putenv	UNIX	put a string into the environment
rawcon	AmigaDOS	set or unset raw console input
stacksize	SAS/C	get the current stack size
stackused	SAS/C	get the amount of stack in use
system	ANSI	call the system command processor

Directory Functions

Function	Portability	Description
chdir	UNIX	change the current directory
closedir	UNIX	terminate the directory operation
dfind	AmigaDOS	find a directory entry
dnext	AmigaDOS	find the next directory entry
findpath	AmigaDOS	locate a file in the current path
getcd	AmigaDOS	get the current directory
getcwd	UNIX	get the current working directory
getfnl	SAS/C	get a filename list
getpath	AmigaDOS	get the path for a specific directory or file
mkdir	UNIX	make a new directory
opendir	UNIX	initiate a directory operation
readdir	UNIX	read a directory element
rmdir	UNIX	remove a directory

(continued)

	Function	Portability	Description
	seekdir	UNIX	reposition a directory operation
	rewinddir	UNIX	reset to the start of the directory
	telldir	UNIX	get the directory position
Localization Functions	Function	Portability	Description
	localeconv	ANSI	return information on locale formatting conventions
	readlocale	SAS/C	read and initialize a new locale
	setlocale	ANSI	set locale information for a program
	strcoll	ANSI	compare strings based on locale
	strxfrm	ANSI	transform a string
Multibyte Character Functions	Function	Portability	Description
	mblen	ANSI	determine the length of a multibyte character
	mbstowcs	ANSI	convert a multibyte string to a wide-character string
	mbtowc	ANSI	map a multibyte character to a wide character
	wcstombs	ANSI	convert a wide-character string to a multibyte string
	wctomb	ANSI	map a wide character to a multibyte character
Functions the User Can Replace But Not Call	Function	Portability	Description
	_CXFERR	SAS/C	low-level floating-point error
	_CXOVF	SAS/C	default stack-overflow handler

Built-in Functions	Function	Portability	Description
	strcpy	ANSI	copy one string to another
	strlen	ANSI	measure the length of a string
	strcmp	ANSI	compare strings
	memcmp	ANSI	compare two memory blocks
	memcpy	ANSI	copy a memory block
	memset	ANSI	set a memory block to a value
	abs	ANSI	absolute value
	max	UNIX	compute the maximum of two values
	min	UNIX	compute the minimum of two values
	__emit	SAS/C	emit 68000 instruction word
	getreg	SAS/C	get 68000-specific registers
	putreg	SAS/C	set up 68000-specific registers
	geta4	AmigaDOS	establish addressability to the global data area

abort Abort the current process

Synopsis `#include <stdlib.h>`

```
void abort(void);
```

Description This function aborts the current process and returns a completion code of 3 to the parent process. Also, the message **Abnormal program termination** is sent to `stderr`. Level 2 I/O buffers are flushed.

Portability ANSI

Example

```
/*
 * Do your own memory allocation
 */
#include <stdio.h>
#include <stdlib.h>

void *gmem(size_t size)
{
    void *p;

    if ((p=malloc(size))==NULL)
    {
        abort();
    }
    return(p);
}

void main(void)
{
    void *p;

    p = gmem(4096);
    if (p)
    {
        free(p);
    }
}
```

See Also `exit`, `__exit`, `raise`, `_XCEXIT`

abs Absolute value

Synopsis `#include <stdlib.h>`

```
ax = abs(x);
```

```
type x;
```

```
type ax;
```

Description This macro computes the absolute value of the various numeric data types. It accepts any data type as its argument and generates in-line code to perform the conversion. The definition is

```
#define abs(x) ((x)<0?-(x):(x))
```

To minimize code size, you can use the **fabs**, **iabs**, or **labs** function instead. Choose the function that corresponds to the data type being converted.

Portability ANSI

Returns The return value is the absolute value of the argument.

See Also **fabs**, **iabs**, **labs**

access Check file accessibility

Synopsis `#include <stdio.h>`

```
ret = access(name,mode);

int ret;           /* return code */
const char *name;  /* file name */
int mode;          /* access mode */
```

Description This function checks if a file is accessible in the way specified by the **mode** parameter. The following table shows the modes, which follow the UNIX format, accepted by this function.

Name	Value	Meaning
R_OK	4	check if the file is readable
W_OK	2	check if the file is writable
X_OK	1	check if the file is executable
F_OK	0	check if the file exists

You can simultaneously check for more than one attribute by combining these flags using the logical OR operator.

This function is not available if the `__STRICT_ANSI` flag has been defined.

Portability UNIX

Returns A return value of 0 indicates that access is allowed. If access is denied or the file cannot be found, `-1` is returned. Additional error information can then be found in the external integers `errno` and `_OSERR`.

Example

```
/*
 * Check to see if the file s:myconfig
 * exists and is writable.
 */
#include <stdio.h>
```

access Check file accessibility
(continued)

```
void main(void)
{
    if (access("s:myconfig",F_OK|W_OK)==0)
    {
        printf("yes\n");
    }

    else
    {
        printf("no\n");
        exit(EXIT_FAILURE);
    }
}
```

See Also `chmod`, `errno`, `_OSERR`

acos Arccosine function

Synopsis `#include <math.h>`

```
    r = acos(x);
```

```
    double r;  /* result */
    double x;  /* angle */
```

Description The **acos** routine computes the arccosine of **x** and returns the angular value expressed in radians. The result is constrained from 0 to π . The argument **x** must be between -1.0 and 1.0 , inclusive, or a **DOMAIN** error will be raised.

Portability ANSI

Returns This function returns the arccosine of the angle expressed in radians.

See Also **cos**, **__matherr**

argopt Get options from an argument list

Synopsis `#include <stdlib.h>`

```
optd = argopt(argc,argv,opts,argn,optc);
```

```
char *optd;           /* option data pointer          */
int argc;             /* argument count            */
const char *argv[];   /* argument vector           */
const char *opts;     /* options expecting data    */
int *argn;            /* next argument number (changed) */
char *optc;           /* option character (changed)  */
```

Description This function examines an argument list to find the next option argument, using the conventions similar to those of the UNIX shell command processor. These conventions follow:

- An option is an argument that begins with a slash (/) or a dash (a minus sign) and appears between the command verb (`argv[0]`) and the first non-option argument. This function recognizes either a slash or a dash because these characters are used by other systems.
- The character immediately following the dash is called the *option character*, and it may be followed by a character string known as the *option data*.
- If the option character appears in the `opts` string, then the data can be separated from the character by white space. In effect, this means that the data might be in the next `argv` entry if it does not follow the option character in the current entry.
- A dash or slash followed by a blank or a dash indicates the end of the options.

Each time **argopt** is called, it finds the next option in the argument array and updates the integer referenced by the `argn` parameter. On the first call, you should set this integer to 1, since `argv[0]` points to the command verb. The `argc` and `argv` items are normally the same as those passed to your main program, and they are not changed as a result of the **argopt** calls. The option character is returned in the byte referenced by the `optc` parameter, and the function returns a pointer to the option data string or to a `NULL` byte. If the next entry in the `argv` vector is not an option, then the function returns a `NULL` pointer.

The `opts` item provides some flexibility in the way the option data are handled. If the `opts` parameter points to an empty string, then any option data must immediately follow the option character. However, if the

argopt Get options from an argument list
(continued)

opts parameter is not empty, then it lists the option characters that always have data. For those characters, the data can be preceded by white space in the command line. What this actually means is that the **argopt** function will look at the next entry in the **argv** vector if the option character is not followed by a data string. If that next entry does not begin with a dash, then it is taken as the option data. See the examples below for clarification.

This function is not available if the **__STRICT_ANSI** flag has been defined.

Portability SAS/C

Returns If the next argument is not an option, the function returns a **NULL** pointer. Otherwise, it returns a pointer to the option data, which will be an empty string if there were no data. Also, if an option was found, the character is placed into the byte referenced by the **optc** parameter, and the **argn** parameter is adjusted to index the next entry in the **argv** vector.

Example

```

/*
 * Assume that this program is invoked with the following
 * command line:
 *
 *   myprog -x -ypdq -z -g moo -g - blah
 *
 * The output will then be:
 *
 *   Option: x Data:
 *   Option: y Data: pdq
 *   Option: z Data:
 *   Option: m Data: oo
 *   Option: g Data:
 *   Arg[8]: blah
 */
#include <stdio.h>
#include <stdlib.h>

char opts[] = "gx";

```

argopt Get options from an argument list
(continued)

```
void main(int argc, char *argv[])
{
    char option,*odata;
    int next;

    next = 1;
    while((odata = argopt(argc,argv,opts,&next,&option))
        != NULL)
    {
        printf("Option: %c, Data: %s\n",option,odata);
    }
    for (; next < argc; next++)
    {
        printf("Arg[%d]: %s\n",next,argv[next]);
    }
}
```

See Also main

asctime Generate an ASCII time string

Synopsis `#include <time.h>`

```
s = asctime(t);

char *s;           /*points to time string */
const struct tm *t; /*points to time structure */
```

Description This function converts a time structure into an ASCII string. The time structure argument `t` is usually returned by the `gmtime` or `localtime` function.

Portability ANSI

Returns This function returns an ASCII string of exactly 26 characters having the form:

```
"DDD MMM dd hh:mm:ss YYYY\n\0"
```

DDD is the day of the week, MMM is the month, dd is the day of the month, hh:mm:ss is the hour:minute:seconds, and YYYY is the year. An example is

```
"Wed Sep 04 15:13:22 1985\n\0"
```

The time pointer returned by the function refers to a static data area that is shared by both the `ctime` and `asctime` functions.

Example `#include <stdio.h>`
`#include <time.h>`

```
void main(void)
{
    struct tm *tp;
    long t;

    time(&t);
    tp = localtime(&t);
    printf("Current time is %s\n",asctime(tp));
}
```


asctime Generate an ASCII time string
(continued)

See Also `ctime`, `gmtime`, `localtime`, `strftime`

asin Arcsine function

Synopsis `#include <math.h>`

```
r = asin(x);

double r;    /* result*/
double x;    /* angle */
```

Description The **asin** routine computes the arcsine and returns an angular value expressed in radians. The result is constrained as $-\pi/2$ to $\pi/2$. The argument must be between -1.0 and 1.0 , inclusive, or a **DOMAIN** error will be raised.

Portability ANSI

Returns This function returns the arcsine of the argument expressed in radians.

See Also `__matherr`, `sin`

assert Assert program validity

Synopsis `#include <assert.h>`

```
assert(x);
__assert(x, file, line);

int x;
const char *file;    /* source file name */
int *line;           /* source line number */
```

Description The **assert** macro tests an expression **x** for validity (nonzero value). If a condition in your program is false (0), the **assert** macro is a quick way to abort the program and print an error message to **stderr**. If the expression is 0 (false), then the macro calls the **__assert** function with the expression in text form plus the source filename (as defined in the **__FILE__** macro) and line number (as defined in the **__LINE__** macro), also as text strings. The default version of the **__assert** function prints a message to **stderr** and aborts with an exit code of 1.

To define the macro, include the **assert.h** header file in your program. The **assert.h** file contains two versions of the macro, a null version and the normal code-generating version. Using the null version allows you to strip the assertion code from your program without removing the **assert** calls. To use the null version, define the symbol **NDEBUG** in one of your header files. If you define **NDEBUG** in one of your header files, the header file containing the **NDEBUG** definition must be included before the **assert.h** file. If the symbol **NDEBUG** is defined, the null version of the macro is used. If the symbol **NDEBUG** is not defined, the normal code-generating version applies.

Portability ANSI

Example

```
#include <stdio.h>
#include <assert.h>

/*
 * Make sure integer x is positive
 */

void posttest(int x)
{
    assert(x >= 0);
}
```

assert Assert program validity
(continued)

```
void main(void)
{
    posttest(5);
    printf("5 is a positive integer\n");
}
```

astcsma AmigaDOS string pattern match (anchored)

Synopsis `#include <string.h>`

```
length = astcsma(s,p);

int length;      /* length of matching string */
const char *s;   /* string being scanned */
const char *p;   /* pattern string */
```

Description The function **astcsma** performs a simple anchored pattern match of the type used by AmigaDOS. The pattern must match at the beginning of the supplied string.

This function is not available if the **__STRICT_ANSI** flag has been defined.

Portability AmigaDOS

Returns This function returns the length of the matching string or 0 if there was no match.

Example

```
/*
 * Show all files matching the pattern "#?.c"
 */
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <types.h>
#include <dir.h>

void main(void)
{
    DIR *dfd;
    struct direct *dptr;

    dfd=opendir(""); /* opens the current directory */
```

astcsma AmigaDOS string pattern match (anchored)
(continued)

```

    if (dfd != NULL)
    {
        while ((dptr=readdir(dfd))!=NULL)
        {
            if (astcsma(dptr->d_name,"#?.c")!=0)
            {
                printf("%s\n",dptr->d_name);
            }
        }
        closedir(dfd);
    }
}

```

See Also stcpm, stcpma, stcsma

atan Arctangent function*Synopsis* `#include <math.h>`

```

r = atan(x);

double r;    /* result */
double x;    /* angle */

```

Description The **atan** routine computes the arctangent of **x** and returns an angular value expressed in radians. The result is constrained as $-\pi/2$ to $\pi/2$.

Since the tangent becomes very large for angles close to $\pi/2$, the **atan2** function is often used to avoid computations with large numbers that might easily overflow. With the **atan2** function, you can express the large tangent value as a quotient of two more reasonable numbers.

Portability ANSI

Returns This function returns the arctangent of the argument expressed in radians.

See Also **atan2**, **__matherr**, **tan**

atan2 Arctangent of x/y

Synopsis `#include <math.h>`

```
r = atan2(x,y);

double r;    /* result; */
double x,y;  /* angle */
```

Description The `atan2` routine computes the arctangent of x/y and returns an angular value expressed in radians. The result is constrained as $-\pi/2$ to $\pi/2$. You can express the large tangent value as a quotient of two more reasonable numbers.

Since the tangent becomes very large for angles close to $\pi/2$, the `atan2` function is often used to avoid computations with large numbers that might easily overflow.

Portability ANSI

Returns This function returns the arctangent of the argument expressed in radians.

See Also `atan`, `__matherr`, `tan`

atexit Set an exit trap

Synopsis `#include <stdlib.h>`

```
error = atexit(func);
```

```
int error;                /* zero for success      */
void (*func)(void);       /* trap function pointer */
```

Description This function registers an exit trap, which will be called when the program exits. There is no limit to the number of exit functions you can have. Exit functions are called in last in, first out (LIFO) order.

After you establish an exit routine, you cannot prevent it from being called. All functions will be called, in reverse order of their registration, when the program exits.

At the time the exit traps are called, all files opened with the `open` or `fopen` function are still open, unless specifically closed by your code. All memory allocated with the `malloc` function and other ANSI memory-allocation functions is still allocated unless specifically freed by your code. After the exit traps have been called, these resources are freed.

Portability ANSI

Returns A zero is returned for success and a nonzero value is returned if an error is encountered.

See Also `exit`

atof Convert an ASCII string to a floating-point number

Synopsis `#include <stdlib.h>`

```
d = atof(p);

double d;          /* floating point result */
const char *p;     /* input string pointer */
```

Description This function converts an ASCII input string into a double-precision floating-point number. The string can contain leading white space and a plus or minus sign, followed by a valid floating-point number in normal or scientific notation. If scientific notation is used, there can be no white space between the number and the exponent. For example, the following is a valid number in scientific notation:

```
123.456e-53
```

Portability ANSI

Returns This function returns the double-precision floating-point equivalent of the ASCII string.

Example

```
/*
 * This program tests the atof function.
 */
#include <stdio.h>
#include <math.h>

void main(void)
{
    char buff[80];
    double d;

    while(1)
    {
        printf("\nEnter a number: ");
        if (fgets(buff, sizeof(buff), stdin) == NULL)
        {
            break;
        }
    }
}
```

atof Convert an ASCII string to a floating-point number
(continued)

```
        if (buff[0] == '\\0')
        {
            break;
        }
        d = atof(buff);
        printf("%e\\n",d);
    }
    printf("\\n");
}
```

See Also `atoi`, `atol`, `strtol`, `strtol`, `strtod`

atoi Convert an ASCII string to an integer

Synopsis `#include <stdlib.h>`

```
x = atoi(s);  
  
int x;  
const char *s;
```

Description This function converts an ASCII string into a normal integer. The string must have the form:

`[whitespace][sign]digits`

Whitespace indicates optional leading white space, *sign* indicates an optional + or – sign character, and *digits* is a contiguous string of digit characters. Once the digit portion is reached, the conversion continues until a nondigit character is reached. No check is made for integer overflow.

Portability ANSI

Returns This function returns the integer equivalent of the ASCII string.

See Also `atof, atol, strtod, strtol`

atol Convert an ASCII string to a long integer

Synopsis `#include <stdlib.h>`

```
y = atol(s);

long int y;
const char *s;
```

Description This function converts ASCII strings into long integers. The string must have the form:

[whitespace][sign]digits

Whitespace indicates optional leading white space, *sign* indicates an optional + or − sign character, and *digits* is a contiguous string of digit characters. Once the digit portion is reached, the conversion continues until a nondigit character is reached. No check is made for integer overflow.

Portability ANSI

Returns This function returns the long integer equivalent of the ASCII string.

See Also `atof, atoi, strtod, strtol`

bldmem Build a memory pool of a specified size

Synopsis `#include <stdlib.h>`

```
i=bldmem(n);

int n;      /* number of 1K-byte blocks in pool */
int i;      /* -1 for failure */
```

Description The **bldmem** function uses the **sbrk** function to get up to **n** contiguous 1 megabyte blocks of memory for the pool. If **n** is 0, the pool is initialized, but no memory is allocated.

This function is not available if the `__STRICT_ANSI` flag has been defined.

Portability OLD

Returns This function returns a `-1` if the **sbrk** function fails.

See Also **getmem**, **getml**, **rlsmem**, **rlsml**, **sbrk**, **sizmem**

bsearch Perform a binary search*Synopsis* `#include <stdlib.h>`

```

void *bsearch(srch, array, n, size, cmp);

const void *srch;      /* object searched for          */
const void *array;     /* pointer to array to search  */
size_t n;              /* number of members of array  */
size_t size;           /* size of each element        */
int (*cmp)(const void *, const void *);
                        /* pointer to comparison function */

```

Description The **bsearch** function scans a sorted array pointed to by the **array** pointer for a match with a search value addressed by the **srch** pointer. **array** is a pointer to the first element of the array to be scanned. **n** is the number of elements in the block, and **size** is the size of each element in bytes.

The **bsearch** function calls a user-provided comparison function, **cmp**, passing it pointers to the two objects being compared. The first pointer points to the key. The second pointer points to an array element.

The **cmp** function returns the following values:

- a negative integer if the first of the two objects is less than the second
- a positive integer if the first object is greater than the second
- 0 if the two objects are equal.

The array to be searched should be sorted in ascending order.

Portability ANSI

Returns The **bsearch** function returns a pointer to the element that matches the search value. If no match can be found, the **bsearch** function returns a **NULL** pointer.

See Also **qsort**

calloc Allocate and clear memory

Synopsis `#include <stdlib.h>`

```
b = calloc(nelt, esize);

void *b;          /* block pointer      */
size_t nelt;      /* number of elements */
size_t esize;     /* element size       */
```

Description This function is called a *level 3 memory allocation* function because it calls upon level 2 functions. This level is fully compatible with UNIX and with the ANSI Standard.

The `calloc` function uses the `malloc` function to get a level 3 memory block whose size in bytes is given by

```
n = nelts * esize;
```

Then, the block is cleared to zeroes. The `calloc` function returns a `NULL` pointer if the block cannot be allocated. The `calloc` function can only allocate 64 megabytes at a time if you are compiling with short integers.

Note: The level 1 function `rbrk` frees all level 2 and level 3 blocks. You can free the block returned from the `calloc` function with the `free` function.

Portability ANSI

Returns A `NULL` pointer is returned if there is not enough space for the requested block.

Example The following example allocates an array large enough for 100 long integers and initializes the block to zeroes.

```
long *lary;

lary=calloc(100,sizeof(long));
```

The following example allocates an array of data structures.

```
#include <stdio.h>
#include <stdlib.h>
```


calloc Allocate and clear memory
(continued)

```
#define MAXENTRIES 50

struct data
{
    int age;
    char name[30];
    int height;
};

struct data *datap;

void init(void)
{
    if ((datap=calloc(MAXENTRIES,sizeof(struct data)))==NULL)
    {
        abort();
    }
}

void main(void)
{
    init();
    printf("Memory allocated.\n");
    free(datap);
}
```

See Also **free, getmem, malloc, rbrk, realloc, rlsmem, sbrk**

ceil Get floating-point limits

Synopsis `#include <math.h>`

```
x = ceil(y);
```

```
double x,y;
```

Description The `ceil` function returns the smallest whole number not less than the specified real number.

Portability ANSI

Returns The result is a real number.

Example

```
#include <stdio.h>
#include <math.h>

double r;

void main(void)
{
    r = ceil(523.96);    /* r contains 524.0 */
    printf("ceil(523.96) = %lf\n", r);
}
```

See Also `floor`

chdir Change the current directory

Synopsis `#include <stdio.h>`

```
error = chdir(path);

int error;          /* 0 if successful */
const char *path;   /* points to new directory path string */
```

Description This function changes the current directory to the specified path. For AmigaDOS, the path may begin with a drive or volume name and a colon.

This function is not available if the `__STRICT_ANSI` flag has been defined.

Portability UNIX

Returns If the return value is nonzero, then the operation failed. An AmigaDOS error code will be in the external integer `_OSERR`, and a UNIX error code will be in the external integer `errno`.

See Also `errno`, `getcd`, `getcwd`, `mkdir`, `_OSERR`, `rmdir`

chgclk Change the system clock

Synopsis `#include <dos.h>`

```
error = chgclk(clock);

int error;
const unsigned char *clock;
```

Description The **chgclk** function changes the setting of the system clock to the value specified in the array. The array's contents are as follows:

Byte	Contents
clock[0]	day of the week (0 for Sunday)
clock[1]	years since 1980
clock[2]	month (1 to 12)
clock[3]	day (1 to 31)
clock[4]	hour (0 to 23)
clock[5]	minute (0 to 59)
clock[6]	second (0 to 59)
clock[7]	hundredths (0 to 99)

This function changes only the system clock. To make the change permanent, you must use the AmigaDOS command **setclock opt save**.

Portability AmigaDOS

Returns If the array is invalid or the operation failed, the **chgclk** function returns a nonzero value.

See Also **errno**, **getclk**, **_OSERR**

Chk_Abort, chkabort Check for a break character

Synopsis `#include <dos.h>`

`void chkabort(void);`

`void Chk_Abort(void);`

Description These functions force immediate checking for the Control-C and Control-D break characters. Normally, this check only occurs when level 1 I/O is performed. The `chkabort` function provides a mechanism for detecting the break characters at other times, an important function for programs that do little or no I/O processing.

If the break key is detected, the break trap is executed. The break trap is simply a function that gets called whenever the user enters Control-C. If the break trap returns a zero value, control returns to the calling function; otherwise, the program terminates. If the program is invoked from the Workbench, the default handler displays a requester allowing the user to abort the program; otherwise, the default handler prints a message to the Command Line Interpreter (CLI) and aborts.

The `chkabort` function ignores the break characters Control-E and Control-F. The `chkabort` function may be replaced with calls to the `onbreak` or `signal` functions.

Portability AmigaDOS

See Also `onbreak`, `signal`

chkml Check for the largest memory block

Synopsis `#include <stdlib.h>`

```
size = chkml(void);
```

```
long size;
```

Description This function returns the size, in bytes, of the largest block that is currently available in the level 2 memory pool without calling the operating system to supply additional heap space.

 This function is not available if the `__STRICT_ANSI` flag has been defined.

Portability SAS/C

Returns The return value is the size of the largest level 2 block.

See Also `getmem`, `getml`, `rlsmem`, `rlsml`, `sizmem`

chkufb Check a level 1 file handle

Synopsis `#include <ios1.h>`

```
ufb = chkufb(fh);
```

```
struct UFB *ufb; /* pointer to UNIX file block */
int fh;          /* file handle */
```

Description This function checks if a file handle is currently associated with a level 1 file. Normally, it is used internally by the **open**, **close**, **read**, **write**, **lseek** and **tell** functions. The UFB structure is defined in the header file `ios1.h`. For AmigaDOS, this structure has the following form:

```
struct UFB
{
    struct UFB *ufbnxt; /* next UFB */
    int ufbflg;         /* flags */
    long ufbfh;         /* file handle */
    char *ufbfname;     /* file name */
}
```

The structure pointer **ufbnxt** points to the next entry in the linked list containing all currently open level 1 files. The integer **ufbflg** contains the mode flags specified in the call to the **open** function. The long integer **ufbfh** contains the file handle. **ufbfname** is a pointer to the character string containing the filename.

The external data name `__ufbs` refers to a linked list of UFB structures, and the external integer `__nufbs` indicates how many structures are in the linked list.

Portability SAS/C

Returns If no UFB is currently attached to the file handle, a **NULL** pointer is returned.

See Also `__nufbs`, `__ufbs`

chmod Change a file's protection mode

Synopsis

```
#include <stdio.h>
#include <fcntl.h>

error = chmod(name,mode);

int error;          /* error code */
const char *name;   /* file name */
int mode;           /* protection mode */
```

Description This function changes a file's protection mode. It is compatible with UNIX, although AmigaDOS also provides separate delete protection for each file. The mode argument should be formed combining any of the following symbols, which are defined in the file `fcntl.h`, using the logical OR operator:

Value	Meaning
S_IWRITE	write permission
S_IREAD	read permission
S_IEXECUTE	execute permission
S_IDELETE	delete permission
S_IPURE	set the bit
S_IARCHIVE	indicate the file has been backed up
S_ISCRIPT	indicate the file is a system script

This function is not available if the `__STRICT_ANSI` flag has been defined.

Portability UNIX

Returns If the operation is successful, the function returns 0. Otherwise, it returns `-1` and places error information in the external integers `errno` and `_OSERR`.

chmod Change a file's protection mode
(continued)

Example

```
/*
 * This example changes file "xyz/pdq.x" so it
 * can be read and written.
 */
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>

void main(void)
{
    if (chmod("xyz/pdq.x", S_IWRITE | S_IREAD))
    {
        perror("Change mode");
        exit(EXIT_FAILURE);
    }
}
```

See Also **access**

clearerr Clear a level 2 I/O error flag

Synopsis `#include <stdio.h>`

`void clearerr(fp);`

`FILE *fp;            /* file pointer */`

Description This macro clears the error flag and end-of-file flag of the file pointed to by the `fp` pointer. Once set, the error flag forces an end-of-file value to be returned any time the file is accessed until the flag is reset.

Portability ANSI

See Also `clrerr`, `ferror`

clock Measure program processor time

Synopsis `#include <time.h>`

```
x = clock(void);
```

```
clock_t x;
```

Description The `clock` function returns the amount of processor time used by the program, relative to the base point. The *base point* for the `clock` function is the value returned by the first call to `clock`. The first call to the `clock` function always returns 0.0.

You can use the `clock` function to determine the amount of processor time used by calling the `clock` function twice in the same program and calculating the difference between the values returned.

The value returned is of type `clock_t`. The value returned is in fractions of a second, where a value of `CLOCKS_PER_SEC` represents one second of processor time. (`clock_t` and `CLOCKS_PER_SEC` are defined in the file `time.h`.) In this implementation, `clock_t` is defined as an unsigned long integer and `CLOCKS_PER_SEC` is 1,000.

The value returned by the `clock` function is of relatively low accuracy and may depend on the extent of other system activity. Values returned by the `clock` function are likely to be inconsistent from one execution of a program to another.

Portability ANSI

Returns The `clock` function returns the number of seconds since the base time. If an accurate value cannot be returned, `(clock_t)-1` is returned.

Example

```
/*
 * This example determines the processor time
 * required to compute 1000 logarithms.
 */

#include <time.h>
#include <math.h>
#include <stdio.h>

void main(void)
{
    clock_t start, end;
    double index;
```

clock Measure program processor time
(continued)

```
        /* Time used before start of computation */
start = clock(void);

for (index = 1.0; index <= 1000.0; ++index )
    (void)log (index);

end = clock(void);

        /* Calculate the difference in time and print */
printf("Processor time used = %g\n",
       (end - start) / CLOCKS_PER_SEC);
}
```

See Also `system`, `time`

close Close a level 1 file

Synopsis `#include <fcntl.h>`

```
error = close(fh);
```

```
int error; /* non-zero if error */
int fh;    /* file handle */
```

Description This function closes a level 1 file that was previously opened with the **open** function. Any pending output is completed and the file directory is updated.

All level 1 files are automatically closed when your program terminates, but it is good programming practice to close a file when you are finished with it. One reason for doing this is to free up the operating system resources (for example, control blocks and buffers) that are allocated for the file while it remains open.

Portability UNIX

Returns The function returns 0 if it is successful. Otherwise, it returns `-1` and places additional error information into the external integers `errno` and `_OSERR`.

Example See the **open** function.

See Also `_dclose`, `fclose`, `open`

closedir Terminate the directory operation

Synopsis `#include <dir.h>`

```
void closedir(dfd)
DIR *dfd;          /* directory file descriptor */
```

Description This function closes a directory previously opened with the **opendir** function. All memory and systems locks associated with the directory file descriptor are freed. You must call the **closedir** function for each descriptor opened with the **opendir** function to reclaim the memory and the lock associated with the descriptor.

Portability UNIX

See Also **opendir**, **readdir**, **seekdir**, **telldir**

clrerr Clear a level 2 I/O error flag

Synopsis `#include <stdio.h>`

```
void clrerr(fp);
```

```
FILE *fp;          /* file pointer */
```

Description This macro clears the error flag associated with the specified level 2 file. Once set, the error flag forces an end-of-file value to be returned any time the file is accessed until the flag is reset. This function duplicates the ANSI `clearerr` macro and is provided for compatibility.

This function is not available if the `__STRICT_ANSI` flag has been defined.

Portability UNIX

See Also `clearerr`, `ferror`, `fopen`

cos Cosine function

Synopsis `#include <math.h>`

```
r = cos(x);  
  
double r; /* result */  
double x; /* angle */
```

Description The `cos` function computes the cosine of angles expressed in radians.

Portability ANSI

Returns This function returns the cosine of the argument.

See Also `__matherr`

cosh Hyperbolic cosine function

Synopsis `#include <math.h>`

```
r = cosh(x);

double r;    /* result */
double x;    /* angle */
```

Description The **cosh** function computes the hyperbolic cosine of an angle. A range error occurs if the hyperbolic cosine is too large to be represented by a double.

Portability ANSI

Returns This function returns the hyperbolic cosine of the argument expressed in radians.

See Also **exp**, **__matherr**

cot Cotangent function

Synopsis `#include <math.h>`

```
r = cot(x);

double r;    /* result */
double x;    /* angle */
```

Description The `cot` function computes the cotangent of an angle measured in radians.

This function is not available if the `__STRICT_ANSI` flag has been defined.

Portability UNIX

Returns This function returns the cotangent of the argument expressed in radians.

See Also `__matherr`

creat Create a level 1 file

Synopsis `#include <fcntl.h>`

```
fh = creat(name,prot);

int fh;           /* file handle */
const char *name; /* file name */
int prot;         /* protection mode */
```

Description This function is exactly the same as calling the **open** function in the following way:

```
open(name,O_WRONLY | O_TRUNC | O_CREAT , prot)
```

The file is created if it does not exist and truncated if it does exist. Then, it is opened for writing. The protection mode can be any of the following:

Value	Meaning
S_IWRITE	write permission
S_IREAD	read permission
S_IWRITE S_IREAD	write and read permission
0	write and read permission

In the current AmigaDOS implementation, the protection mode is ignored.

Portability UNIX

Returns If the operation is successful, the function returns a file handle, which is an integer equal to or greater than 0. Otherwise, it returns `-1` and places error information in the external integers **errno** and **_OSERR**.

See Also **chmod**, **close**, **errno**, **open**, **_OSERR**

ctime Convert a time value to a string

Synopsis `#include <time.h>`

```
s = ctime(t);
```

```
char *s;           /* points to time string */
const time_t *t;   /* points to time value  */
```

Description This function converts a `time_t` time value to a local time in the form of an ASCII string.

The `ctime` function is equivalent to the following:

```
asctime(localtime(t));
```

Portability ANSI

Returns This function returns an ASCII string of exactly 26 characters having the form:

```
"DDD MMM dd hh:mm:ss YYYY\n\0"
```

DDD is the day of the week, MMM is the month, dd is the day of the month, hh:mm:ss is the hour:minute:seconds and YYYY is the year. An example is

```
"Wed Sep 04 15:13:22 1985\n\0"
```

The time pointer returned by the function refers to a static data area that is shared by both the `ctime` and `asctime` functions.

ctime Convert a time value to a string
(continued)

Example

```
#include <time.h>
#include <stdio.h>

void main(void)
{
    time_t t;

    time(&t);
    printf("Current time is %s", ctime(&t));
}
```

See Also asctime, gmtime, localtime, strftime, time

__CXFERR Low-level floating-point error

Synopsis `#include <math.h>`

```
void __CXFERR(code);
int code;
```

Description This function is called when an error is detected by one of the low-level floating-point routines, such as the arithmetic operations. Higher-level routines, such as the trigonometric functions, use the `__matherr` function.

You cannot call this function. However, you can replace this error trap with an application-dependent routine, as long as it stores the error code in the global integer `__FPERR`. Some of the math functions check the `__FPERR` integer to see if low-level errors occurred. To replace this function, include a prototype as well as the error trap function in your source code. The function and prototype should be named `__CXFERR`.

The error code passed to the `__CXFERR` function indicates the type of floating-point anomaly that occurred, as follows:

Symbol	Value	Meaning
<code>__FPEUND</code>	1	underflow
<code>__FPEOVF</code>	2	overflow
<code>__FPEZDV</code>	3	divide by zero
<code>__FPENAN</code>	4	not a number
<code>__FPECOM</code>	5	not comparable

If the `__STRICT_ANSI` flag has not been defined, equivalent names without the leading underscore are also defined for compatibility with previous releases of the compiler.

These codes are defined in the file `math.h`.

This routine was named `CXFERR` in Version 5 and previous versions.

Portability SAS/C

See Also `__matherr`

_CXOVF Default stack-overflow handler

Synopsis `_CXOVF(void);`

Description This function is the default stack-overflow handler. When called, it puts up a requester indicating the stack-overflow state for the current program. After the user clicks on the requester, this routine terminates the program.

This function is automatically invoked by the stack-overflow detection code. You cannot call this function. However, you can replace this function with your own replacement function. To replace the `_CXOVF` function, include a prototype as well as the function in your source code. The function and prototype should be named `_CXOVF`.

Sample source code is provided in the source directory.

This routine was named `cxovf` in Version 5 and previous versions.

Portability SAS/C

Example

```

/*
 * This is a replacement overflow handler that simply
 * restarts the program at a safe known spot.
 */
#include <setjmp.h>

extern jmp_buf safe_spot;
void __stdargs _CXOVF(void)
{
    longjmp(safe_spot, 1);
}

```

datecmp Compare two AmigaDOS dates

Synopsis `#include <dos.h>`

```
status = datecmp(d1, d2);

int status;          /* result of comparison: */
                    /* 0 if equal */
                    /* -1 if d1 > d2 */
                    /* 1 if d1 < d2 */
const struct DateStamp *d1; /* first date to compare */
const struct DateStamp *d2; /* second string to compare */
```

Description The **datecmp** function compares two date vectors (three long words) to see if they are equal. To be equal, all of the days, minutes, and ticks must match. If they are unequal, the **datecmp** function returns an appropriate value.

Portability AmigaDOS

Returns If the dates match as described above, then 0 is returned. If the first date is greater than the second date, then -1 is returned. Otherwise, 1 is returned.

See Also **DateStamp** (*The AmigaDOS Manual, 3rd Edition*)

__dclose Close an AmigaDOS file

Synopsis `#include <dos.h>`

```
error = __dclose(fh);

int error;    /* 0 for success, -1 for error */
long fh;     /* file handle                */
```

Description This function closes an AmigaDOS file that was opened with the `__dcreat`, `__dcreatx`, or `__dopen` function. AmigaDOS will not automatically close these files for you. One reason for closing files is to free up the operating system resources (control blocks and buffers) which are allocated for the file while it remains open.

This function is obsolete and provided for compatibility only. Use the AmigaDOS `close` function instead.

Portability OLD

Returns If the operation is successful, the function returns 0. Otherwise, it returns `-1` and places error information in the external integers `errno` and `_OSERR`.

See Also `close` (*The AmigaDOS Manual, 3rd Edition*), `__dcreat`, `__dcreatx`, `__dread`, `__dseek`, `__dwrite`, `errno`, `_OSERR`,

`_dcreat` Create an AmigaDOS file

Synopsis `#include <dos.h>`

```
fh = _dcreat(name,fatt);

long fh;           /* file handle (-1 for error) */
const char *name;  /* file name */
int fatt;          /* file attribute */
```

Description This function creates and opens an AmigaDOS file, returning the file handle. The `_dcreat` operation truncates the file if it already exists or creates the file if it does not exist.

The file attribute argument is accepted to provide source-level compatibility with other systems, but is ignored under AmigaDOS.

This function is obsolete and provided for compatibility only. Use the AmigaDOS `Open` function instead.

Portability OLD

Returns If the operation is successful, the function returns a file handle. Otherwise, it returns `-1` and places error information in the external integers `errno` and `_OSERR`.

See Also `_dclose`, `_dcreatx`, `_dread`, `_dseek`, `_dwrite`, `errno`, `Open` (*The AmigaDOS Manual, 3rd Edition*), `_OSERR`,

`_dcreatx` Create a new AmigaDOS file

Synopsis `#include <dos.h>`

```
fh = _dcreatx(name, fatt);

long fh;           /*file handle (-1 for error) */
const char *name;  /*file name */
int fatt;          /*file attribute */
```

Description This function creates and opens an AmigaDOS file, returning the file handle. The `_dcreatx` operation fails if the file already exists, or creates the file if it does not exist.

The file attribute argument is accepted to provide source-level compatibility with other systems, but is ignored under AmigaDOS.

This function is obsolete and provided for compatibility only. Use the AmigaDOS `Open` function instead.

Portability OLD

Returns If the operation is successful, the function returns a file handle. Otherwise, it returns `-1` and places error information in the external integers `errno` and `_OSERR`.

See Also `_dclose`, `_dcreat`, `_dread`, `_dseek`, `_dwrite`, `errno`, `Open` (*The AmigaDOS Manual, 3rd Edition*), `_OSERR`,

dfind Find a directory entry

Synopsis `#include <dos.h>`

```
error = dfind(info,name,attr);

int error;                      /* 0 if successful */
struct FileInfoBlock *info; /* file information area */
const char *name;              /* file name or pattern */
int attr;                      /* file attribute bits */
```

Description The **dfind** function searches a directory for the first entry that matches the specified filename or filename pattern.

The **name** argument must be a null-terminated string specifying the drive, path, and name of the desired file. The drive and path can be omitted, in which case, the current directory will be searched. You can use the AmigaDOS wildcard characters as defined in the *AmigaDOS Users Manual* for pattern matching in the name portion. For example, **xy#?.b** locates files in the current directory that begin with **xy** and have **b** as their extension. Setting the external integer location **msflag** to a nonzero value causes all subsequent calls to the **dfind** function to use the MS-DOS ***** and **?** characters for pattern matching in the name portion instead of the AmigaDOS characters **#** and **?**.

The **attr** argument specifies which file types are to be included in the search. If **attr** is 0, the search will include only normal files. Otherwise, the search will also include directories.

The **info** argument points to a file information structure as defined in the **dos.h** header file. For AmigaDOS, this is the same as the AmigaDOS **FileInfoBlock** structure.

```
struct FileInfoBlock {
    long    fib_DiskKey;
    long    fib_DirEntryType;
    char    fib_FileName[108];
    long    fib_Protection;
    long    fib_EntryType;
    long    fib_Size;
    long    fib_NumBlocks;
    struct DateStamp fib_Date;
    char    fib_Comment[116];
};
```

info is a pointer to a file information block that must be allocated on a 4-byte (long word) boundary by the calling program. A common error

dfind Find a directory entry
(continued)

is failing to allocate the structure before calling the function. You can make sure the structure is long-word aligned by either declaring it with the `__aligned` keyword or by allocating it dynamically with any SAS/C or system allocation function (such as the `malloc` or `AllocMem` function).

Portability AmigaDOS

Returns If the operation is successful, a value of 0 is returned. Otherwise, the return value is `-1`, and you can find additional error information in the external integers `errno` and `_OSERR`.

See Also `dnext`, `errno`, `_OSERR`

difftime Compute the difference between two `time_t` values

Synopsis `#include <time.h>`

```
x = difftime(time2,time1);
```

```
double x;
time_t time1, time2;
```

Description The `difftime` function computes the difference `time2-time1` in seconds, where `time2` and `time1` are values of type `time_t` (such as those returned by the `time` function).

Portability ANSI

Returns The `difftime` function returns the difference between two times in seconds.

Example

```
/*
 * This example demonstrates a method of detecting
 * when time-dependent data needs to be written
 */

#include <time.h>
#include <stdio.h>

void main(void)
{
    time_t last_written; /* Time data was last written */
    FILE *fp;
    int length;

    /*
     * Open and write to a data file. Record when
     * the file is written to
     */
    fp = fopen("datafile","w");
    if (fp == NULL) printf("File wouldn't open\n");
    length = fprintf (fp,"%s\n","Data is written to a file.");

    last_written = time(NULL);

    /* Additional code .... */
```

difftime Compute the difference between two `time_t` values
(continued)

```

/*
 * Write additional data to file if the time since
 * it was last written to is greater than 1 hour
 */
if(difftime ( time(NULL), last_written) > 3600)
    length = fprintf (fp,"%s\n","More data is written.");

fclose(fp);
}

```

See Also `time`

div Compute the quotient and the remainder

Synopsis `#include <stdlib.h>`

```
x = div(y, z);

div_t x;      /* quotient and remainder */
int y;        /* numerator              */
int z;        /* denominator              */
```

Description The `div` function computes the quotient and remainder of the first argument, `y`, divided by the second argument, `z`.

Portability ANSI

Returns The `div` function returns a structure of type `div_t`, which contains both the quotient and remainder. The definition of the `div_t` type is

```
typedef struct {
    int rem;
    int quot;
} div_t;
```

The return value is such that

```
numer = quot * denom + rem
```

The sign of the `rem` value is the same as the sign of the `numer` value.

Example

```
/*
 * This example converts an angle in radians to
 * degrees, minutes, and seconds
 */

#include <stdio.h>
#include <stdlib.h>
#include <math.h>

void main(void)
{
    double rad, angle;
    int deg, min, sec;
    div_t d;
```


div Compute the quotient and the remainder
(continued)

```

rad = 2.414;

    /* Convert angle to seconds and discard fraction */
angle = rad * (180 * 60 * 60)/PI;

sec = angle;
d = div(sec, 60);
sec = d.rem;

d = div(d.quot, 60);
min = d.rem;
deg = d.quot;

printf("%f radians = %d degrees, %d minutes, %d seconds\n",
      rad, deg, min, sec);
}

```

See Also `ldiv`

dnext Find the next directory entry

Synopsis `#include <dos.h>`

```
error = dnext(info);

int error;                /* 0 if successful */
struct FileInfoBlock *info; /* file information area */
```

Description The **dnext** function searches a directory for entries that match the specified filename or filename pattern. You must use the **dfind** function to locate the first matching file. Then, successive calls to the **dnext** function locate additional matching files. Each **dnext** call must be given the file information that was returned on the preceding call to the **dfind** or **dnext** function.

The **info** argument points to a file information structure as defined in the **dos.h** header file. For AmigaDOS, this is the same as the AmigaDOS **FileInfoBlock** structure.

```
struct FileInfoBlock {
    long    fib_DiskKey;
    long    fib_DirEntryType;
    char    fib_FileName[108];
    long    fib_Protection;
    long    fib_EntryType;
    long    fib_Size;
    long    fib_NumBlocks;
    struct DateStamp fib_Date;
    char    fib_Comment[116];
};
```

info is a pointer to a file information block that must be allocated on a 4-byte (long word) boundary by the calling program. A common error is failing to allocate the structure before calling the function. You can make sure the structure is long-word aligned by either declaring it with the **__aligned** keyword or by allocating it dynamically with any SAS/C or system allocation function (such as the **malloc** or **AllocMem** function).

Portability AmigaDOS

Returns If the operation is successful, a value of 0 is returned. Otherwise, the return value is -1, and you can find additional error information in the external integers **errno** and **_OSERR**.

dnnext Find the next directory entry
(continued)

See Also `dfind`, `errno`, `_OSERR`

`_dopen` Open an AmigaDOS file

Synopsis `#include <dos.h>`

```
fh = _dopen(name,mode);

long fh;           /* file handle (-1 for error) */
const char *name;  /* file name */
int mode;          /* access mode */
```

Description This function opens an AmigaDOS file and returns the file handle. The **mode** argument must be a mode supported directly by AmigaDOS and defined in the Amiga header file `dos/dos.h`.

Valid modes are as follows:

MODE_OLDFILE

opens an existing file with read and write access and positions the file pointer at the beginning of the file.

MODE_NEWFILE

opens a new file with read and write access and an exclusive lock. Deletes the old file.

MODE_READWRITE

opens an old file with a shared lock. Creates a new file if it doesn't exist.

This function is obsolete and provided for compatibility only. Use the AmigaDOS **Open** function instead.

Portability OLD

Returns If the operation is successful, the function returns a file handle. Otherwise, it returns `-1` and places error information in the external integers `errno` and `_OSERR`.

See Also `_dclose`, `_dcreat`, `_dcreatx`, `_dread`, `_dseek`, `_dwrite`, `errno`, `open`, `Open` (*The AmigaDOS Manual, 3rd Edition*), `_OSERR`,

dqsort Sort an array of double-precision floating-point numbers

Synopsis `#include <stdlib.h>`

```
void dqsort(da,n);
```

```
double *da; /* pointer to beginning of an array of doubles */
size_t n;   /* number of elements in array */
```

Description The **dqsort** function sorts the specified array of double-precision floating-point numbers using the ACM 271 algorithm, more popularly known as *Quicksort*.

This function is not available if the `__STRICT_ANSI` flag has been defined.

Portability SAS/C

See Also `fqsort`, `lqsort`, `qsort`, `sqsort`, `tqsort`

drand48 Generate a random double-precision floating-point number (internal seed)

Synopsis `#include <math.h>`

```
x = drand48();
```

```
double x;      /* random double */
```

Description This function generates random numbers using the linear congruential algorithm and 48-bit arithmetic. The **drand48** function uses an internal 48-bit storage area for the seed value.

 This function is not available if the **__STRICT_ANSI** flag has been defined.

Returns The **drand48** function returns double-precision floating-point values distributed uniformly over the interval from 0.0 up to but not including 1.0.

Portability UNIX

See Also **erand, jrand48, lcong48, rand, srand, srand48**

__dread Read an AmigaDOS file

Synopsis `#include <dos.h>`

```
count = __dread(fh,buffer,length);

unsigned int count; /* actual number of bytes read */
long fh;           /* file handle */
char *buffer;      /* data buffer */
unsigned int length; /* number of bytes to read */
```

Description This function reads an AmigaDOS file whose handle was returned by the `__dcreat`, `__dcreatx`, or `__dopen` function. Under normal circumstances, the value returned should match the buffer length. If this value is `-1` or greater than the requested length, then some type of error occurred, and you should consult the external integers `errno` and `_OSERR`. If the actual length is less than the requested length when reading, then usually the file is exhausted. Similarly, if the actual length is less than the requested length for a write operation, then usually the device has no more space available. In both of these cases, it is still a good idea to check the external integers `errno` and `_OSERR` in case some malfunction caused the short count.

This function is obsolete and provided for compatibility only. Use the AmigaDOS `Read` function instead.

Portability OLD

Returns If the operation is successful, the function returns the actual number of bytes transferred. Otherwise, it returns `-1` and places error information in the external integers `errno` and `_OSERR`.

See Also `__dclose`, `__dcreat`, `__dcreatx`, `__dseek`, `__dwrite`, `errno`, `_OSERR`, `Read` (*The AmigaDOS Manual, 3rd Edition*)

`_dseek` Reposition an AmigaDOS file

Synopsis `#include <dos.h>`

```
apos = _dseek(fh,rpos,mode);

long apos; /* actual file position */
long fh;   /* file handle */
long rpos; /* relative file position */
int mode;  /* seek mode */
```

Description This function repositions an AmigaDOS file whose handle was returned by the `dcreat`, `dcreatx`, or `dopen` function. The seek mode is the same as for the `lseek` function, as follows:

Mode	Position
0	relative to the beginning of the file
1	relative to the current file location
2	relative to the end of the file

For modes 1 and 2, the `rpos` argument can be positive or negative, but the `apos` argument is always the actual (positive) position relative to the beginning of the file.

This function is obsolete and provided for compatibility only. Use the AmigaDOS `seek` function instead.

Portability OLD

Returns If the operation is successful, the function returns the actual file position, which is a long integer. Otherwise, it returns `-1L` and places error information in the external integers `errno` and `_OSERR`.

See Also `_dclose`, `_dcreat`, `_dcreatx`, `dopen`, `_dread`, `_dwrite`, `errno`, `_OSERR`, `Seek` (*The AmigaDOS Manual, 3rd Edition*)

`_dwrite` Write to an AmigaDOS file

Synopsis `#include <dos.h>`

```
count = _dwrite(fh,buffer,length);

unsigned int count;    /* actual bytes read or written */
long fh;              /* file handle */
char *buffer;         /* data buffer*/
unsigned int length;   /* number of bytes to read or write */
```

Description This function writes an AmigaDOS file whose handle was returned by the `_dcreat`, `_dcreatx` or `_dopen` function. Under normal circumstances, the value returned should match the buffer length. If this value is `-1` or greater than the requested length, then some type of error occurred, and you should consult the external integers `errno` and `_OSERR`. If the actual length is less than the requested length when reading, then usually the file is exhausted. Similarly, if the actual length is less than the requested length for a write operation, then usually the device has no more space available. In both of these cases, it is still a good idea to check the external integers `errno` and `_OSERR` just in case some malfunction caused the short count.

This function is obsolete and provided for compatibility only. Use the AmigaDOS `Write` function instead.

Portability OLD

Returns If the operation is successful, the function returns the actual number of bytes transferred. Otherwise, it returns `-1` and places error information in the external integers `errno` and `_OSERR`.

See Also `_dclose`, `_dcreat`, `_dcreatx`, `_dread`, `_dseek`, `errno`, `_OSERR`, `Write` (*The AmigaDOS Manual, 3rd Edition*)

ecvt Convert a double-precision floating-point number to a string

Synopsis `#include <math.h>`

```
s = ecvt(v,dig,decx,sign);

char *s;    /* string pointer */
double v;   /* floating point value */
int dig;    /* number of digits */
int *decx;  /* pointer to decimal index (returned) */
int *sign;  /* pointer to sign indicator */
```

Description This function converts a floating-point number into an ASCII character string consisting of digits only and terminated by a null character.

The second argument, **dig**, indicates the total number of digits that should be generated. If the floating-point value contains fewer significant digits, zeroes are appended. If there are too many significant digits, the low-order (right-most) digit is rounded.

The **decx** argument points to an integer that receives a value indicating where the decimal point should be placed in the string. For example, an index value of 3 indicates that the decimal point should be placed just after the third character in the string. A value of 0 means that the decimal point is just before the first character. If the index is negative, it indicates the number of zeroes that are between the decimal point and the first character. For example, -3 means that there are three zeroes between the decimal point and the beginning of the string.

The **sign** argument points to an integer that is nonzero if the value **v** is negative.

This function is not available if the `__STRICT_ANSI` flag has been defined.

Portability UNIX

Returns This function returns a pointer to a character string containing the ASCII equivalent of the float argument.

Example

```
#include <stdio.h>
#include <math.h>

void main(void)
{
    int decx, sign;
    char *string;
```

ecvt Convert a double-precision floating-point number to a string
(continued)

```
string = ecvt(3.1415926535, 10, &decx, &sign);  
/* string => "3141592654" */  
/* decx    => 1          */  
/* sign    => 0          */  
printf("string = %s, decx = %d, sign = %d\n",  
       string, decx, sign);  
}
```

See Also `fcvt`

__emit Emit 68000 instruction word

Synopsis `#include <dos.h>`

```
void __emit(inst);
```

```
int inst;
```

Description The built-in function **emit** takes a constant 16-bit integer value corresponding to a 68000 assembly language instruction and inserts it inline with the code. However, it does not check whether the 16-bit value is a valid 68000 instruction. The **__emit** function requires an integer parameter even though the assembly language instruction is only 16 bits long.

► *Caution* **Use this function carefully. Using this function incorrectly can create serious problems.** ▲

Portability SAS/C

Example `emit (0x4180) /* Assembler instruction for chk d0,d0 */`

See Also `getreg, putreg`

erand48 Generate a random double-precision floating-point number (external seed)

Synopsis `#include <math.h>`

```
x = erand48(seed);
```

```
double x;                /* random double */
unsigned short seed[3]; /* seed value (high bits in seed[0]) */
```

Description This function generates random numbers using the linear congruential algorithm and 48-bit arithmetic. The **erand48** function is provided for cases where several seeds are in use at the same time, so you can specify the seed on each function call.

This function is not available if the `__STRICT_ANSI` flag has been defined.

Portability UNIX

Returns The **erand48** function returns double-precision floating-point values distributed uniformly over the interval from 0.0 up to but not including 1.0.

See Also **drand48**, **jrand48**, **lcong48**, **rand**, **srand**, **srand48**

except Call the math error handler function

Synopsis `#include <math.h>`

```
r = except(type,name,arg1,arg2,retval);

double r;           /* actual return value */
int type;           /* error type */
const char *name;    /* math function name */
double arg1;         /* first argument */
double arg2;         /* second argument */
double retval;       /* proposed return value */
```

Description The **except** function is a SAS/C extension to UNIX that simplifies the interface to the `__matherr` function by setting up the exception vector and processing the action code and return value. It is intended to ease the error-handling chore in user-written math functions.

When your math function encounters an error, it should call the **except** function specifying one of the following error types, which are defined in the `math.h` header file:

Symbol	Code	Meaning
<code>__DOMAIN</code>	1	domain error
<code>__SING</code>	2	singularity
<code>__OVERFLOW</code>	3	overflow (number too large)
<code>__UNDERFLOW</code>	4	underflow (number too small)
<code>__TLOSS</code>	5	total loss of significance
<code>__PLOSS</code>	6	partial loss of significance

If the `__STRICT_ANSI` flag has been defined, you must use the alternate entry point `__except`. If the `__STRICT_ANSI` flag has not been defined, equivalent names without the leading underscore are also defined for compatibility with previous releases of the compiler.

Portability SAS/C

except Call the math error handler function
(continued)

Returns The actual return value (a double-precision floating-point value) is passed back.

See Also `__fperr`, `__matherr`

exit Terminate program execution

Synopsis `#include <stdlib.h>`

```
void exit(code);
```

```
int code;
```

Description This function terminates the current program and returns to the operating system. Before exiting, any functions specified in a call to the **atexit** function are called. Next, any open level 1 or level 2 files (opened with the **open** or **fopen** function) are closed. Finally, all level 1 and level 2 memory is released back to the system.

Your program must free any memory allocated with AmigaDOS functions and close any file opened with the AmigaDOS functions.

Portability ANSI

Example

```
/*
 *
 * This example shows how you would abort a program
 * if it is not called with a valid input file name.
 *
 */
#include <stdlib.h>
#include <stdio.h>

void main(int argc, char *argv[])
{
    FILE *f;

    if (argc > 1)
    {
        f = fopen(argv[1], "r");
        if (f == NULL)
        {
            fprintf(stderr,
                    "Can't open file \"%s\"\n",
                    argv[1]);
            exit(EXIT_FAILURE);
        }
        fclose(f);
    }
}
```


exit Terminate program execution
(continued)

```
        else
        {
            fprintf(stderr, "No file specified\n");
            exit(EXIT_FAILURE);
        }
    }
```

See Also `longjmp`

— **__exit** Terminate program execution with no clean-up

Synopsis `#include <stdlib.h>`

```
void __exit(code);
```

```
int code;
```

Description This function terminates the current program immediately and returns control to the parent program. This function does not write output buffers or close level 2 files. Generally, this function is used only in emergency situations when you do not care if some output data are lost.

Files opened with the **open**, **creat**, or **creatx** function are closed.

The **code** parameter is a value from 0 to 255 that gets passed back to the parent. By convention, a value of 0 indicates success.

Portability SAS/C

See Also **exit**

exp Exponential function

Synopsis `#include <math.h>`

```
r = exp(x);
```

```
double r, x;
```

Description The **exp** function raises the natural logarithm base *e* to the *x* power. A range error occurs if *x* is too large.

Portability ANSI

Returns This function returns a double-precision floating-point number containing the calculated exponential.

See Also `log`, `__matherr`

fabs Floating-point or double-precision floating-point absolute value

Synopsis `#include <math.h>`

```
ad = fabs(d);
```

```
double ad,d;
```

Description This function computes the absolute value of either a floating-point number or a double-precision floating-point number. Floating-point arguments are automatically promoted to double-precision floating-point arguments.

Portability ANSI

Returns This function returns a double-precision floating-point number containing the absolute value of the argument.

See Also **abs**

fclose Close a level 2 file

Synopsis `#include <stdio.h>`

```
ret = fclose(fp);
```

```
int ret; /* return code */
```

```
FILE *fp; /* file pointer for file to be closed */
```

Description The **fclose** function completes the processing of a level 2 file and releases all related resources. If the file is in the course of being written, any data which have accumulated in the buffer are written to the file, and the level 1 **__close** function is called for the associated file descriptor. The buffer associated with the file block is freed.

Even though the **fclose** function is automatically called for all open files when your program terminates or calls the **exit** function, it is good programming practice to close your own files explicitly. The last buffer is not written until the **fclose** function is called, and data may be lost if an output file is not properly closed.

Portability ANSI

Returns If successful, the **fclose** function returns 0. This function returns **EOF** to indicate an error. If **EOF** is returned, additional error information can be found in the external integers **errno** and **_OSERR**.

See Also **errno**, **fopen**, **open**, **_OSERR**

fcloseall Close all level 2 files

Synopsis `#include <stdio.h>`

```
num = fcloseall();

int num; /* number of files closed */
```

Description The **fcloseall** function closes all level 2 files that were open and returns that number. However, if an error occurs on any file, the **fcloseall** function continues to close the other files and then returns a value of `-1`.

The **fcloseall** function closes the standard files **stdin**, **stdout**, and **stderr**. Functions such as **printf** and **perror** will fail after you call the **fcloseall** function.

Portability XENIX

Returns If successful, the **fcloseall** function returns the number of files that were closed. This function returns `-1` to indicate an error. If `-1` is returned, additional error information can be found in the external integers **errno** and **_OSERR**.

See Also **errno**, **fopen**, **_OSERR**

fcvt Convert a floating-point number to a string

Synopsis `#include <math.h>`

```
s = fcvt(v,dig,decx,sign);

char *s;    /* string pointer */
double v;   /* floating point value */
int dig;    /* number of digits */
int *decx;  /* pointer to decimal index (returned) */
int *sign;  /* pointer to sign indicator */
```

Description This function converts a floating-point number into an ASCII character string consisting of digits only and terminated by a null character.

The second argument, `dig`, indicates the total number of digits that should be generated. If the floating-point value contains fewer significant digits, zeroes are appended. If there are too many significant digits, the low-order (right-most) digit is rounded.

The `decx` argument points to an integer that receives a value indicating where the decimal point should be placed in the string. For example, an index value of 3 indicates that the decimal point should be placed just after the third character in the string. A value of 0 means that the decimal point is just before the first character. If the index is negative, it indicates the number of zeroes that are between the decimal point and the first character. For example, `-3` means that there are three zeroes between the decimal point and the beginning of the string.

The `sign` argument points to an integer that is nonzero if the value `v` is negative.

This function is not available if the `__STRICT_ANSI` flag has been defined.

Portability UNIX

Returns This function returns a pointer to a string containing the ASCII representation of the `float` argument.

Example

```
#include <stdio.h>
#include <math.h>

void main(void)
{
    int decx, sign;
    char *string;
```

fvct Convert a floating-point number to a string
(continued)

```
string = fvct(3.1415926535,10,&decx,&sign);
/* string => "3141592654" */
/* decx   => 1          */
/* sign   => 0          */
printf("string = %s, decx = %d, sign = %d\n",
       string, decx, sign);
}
```

See Also **ecvt**

fdopen Attach a level 1 file to level 2

Synopsis `#include <stdio.h>`

```
fp = fdopen(fh,mode);

FILE *fp;           /* file pointer */
int fh;             /* file handle */
const char *mode;   /* access mode */
```

Description This function attaches a level 1 file to level 2. In other words, if you have used the **open** function to prepare a file for level 1 I/O processing, you can subsequently use level 2 I/O with that file after calling the **fdopen** function. The file handle is the value returned by the **open** function, and the access mode has the same form as described for the **fopen** function.

Portability UNIX

Returns If the operation is successful, the function returns a file pointer other than **NULL**. If it is not successful, the function returns a **NULL** pointer and places error information in the external integers **errno** and **_OSERR**.

This function is not available if the **__STRICT_ANSI** flag has been defined.

See Also **errno**, **fopen**, **_OSERR**

feof Check for a level 2 end-of-file

Synopsis `#include <stdio.h>`

```
ret = feof(fp);
```

```
int ret;    /* non-zero if condition is true */  
FILE *fp;  /* file pointer */
```

Description This function generates a nonzero value if the specified file pointer is at end-of-file. This function is implemented as a macro. Also, it does not check whether the **fp** argument is a valid file pointer.

Portability ANSI

Returns This function returns a nonzero value if the specified file pointer is at end-of-file. If not, this function returns a 0.

See Also **ferror**

ferror Check for a level 2 error

Synopsis `#include <stdio.h>`

```
ret = ferror(fp);
```

```
int ret;    /* non-zero if condition is true */
FILE *fp;  /* file pointer */
```

Description This function tests the error indicator for the file pointed to by the **fp** argument. This function is implemented as a macro. The **ferror** function does not check whether the **fp** argument is a valid file pointer.

Portability ANSI

Returns The return value is 0 if no error has been set. If an error indicator was set, a nonzero value is returned.

See Also `clearerr`, `feof`

fflush Flush a level 2 output buffer

Synopsis `#include <stdio.h>`

```
ret = fflush(fp);
```

```
int ret;    /* return code */  
FILE *fp;   /* file pointer */
```

Description The `fflush` function flushes the output buffer of the specified level 2 file. That is, it writes the buffer if the file is opened for output and the buffer contains any pending data.

Portability ANSI

Returns If an error occurs, the return value is `-1 (EOF)`. The appropriate error code is placed into the external integer `errno`, and additional information is placed in the external integer `_OSERR`.

See Also `errno`, `fclose`, `fcloseall`, `flushall`, `fopen`, `_OSERR`

fgetc Get a character from a file

Synopsis `#include <stdio.h>`

```
c = fgetc(fp);
```

```
int c;          /* return character or code */
FILE *fp;       /* file pointer */
```

Description This function gets a single character from the specified level 2 file.

In the case of an error, this function returns an **EOF**. To distinguish errors from an end-of-file condition, you should reset the external integer **errno** before calling the function, and analyze its contents when you receive an **EOF** return.

Portability ANSI

Returns If successful, the next input character is returned. Otherwise, the function returns **EOF**, which is defined in the file **stdio.h**. In the event of an **EOF** return, error information can be found in the external integers **errno** and **_OSERR**.

See Also **errno**, **fgetchar**, **fopen**, **fputc**, **getc**, **getch**, **getchar**, **_OSERR**, **ungetc**

fgetchar Get a character from `stdin`

Synopsis `#include <stdio.h>`

```
c = fgetchar(void);
```

```
int c;      /* return character or code */
```

Description This function gets a single character from `stdin`.

In the case of an error, this function returns an **EOF**. To distinguish errors from an end-of-file condition, you should reset the external integer **errno** before calling the function and analyze its contents when you receive an **EOF** return.

This function is identical to the `fgetc(stdin)` function.

This function is not available if the `__STRICT_ANSI` flag has been defined.

Portability XENIX

Returns If successful, the next input character is returned. Otherwise, the function returns **EOF**, which is defined in the `stdio.h` file. In the event of an **EOF** return, error information can be found in the external integers **errno** and **_OSERR**.

See Also **errno**, **fgetc**, **fopen**, **getc**, **getchar**, **_OSERR**

fgetpos Get the current file position

Synopsis `#include <stdio.h>`

```
x = fgetpos(f, pos);
```

```
int x;
FILE *f;
fpos_t *pos;
```

Description The **fgetpos** function determines the current file position for the stream associated with the **FILE** object addressed by the **f** argument, and it stores the file position in the object pointed to by the **pos** argument. This object is of type **fpos_t**, which is defined in the **stdio.h** file. The stored value can be passed to the **fsetpos** function to reposition the file to its position at the time of the call to the **fgetpos** function.

The **fgetpos** function can be used with most types of files, using either text or binary access.

Portability ANSI

Returns If successful, the **fgetpos** function returns 0. If it fails, the **fgetpos** function returns a nonzero value and stores an appropriate error code in the external integer **errno**. See the description of the external integer **errno** in Chapter 6, “Predefined Data Name Reference,” for the list of **errno** values.

Example In the following example, the function **blddbtable** reads a file and builds a table of keys and record addresses. The function **findrec** positions the file to the record with the required key using the **fsetpos** function, and then it reads the record.

```
#include <stdio.h>
#include <string.h>

#define KEYLEN 17
#define DATALEN 500
#define TABSIZE 1000

struct table {
    char keyval[KEYLEN];
    fpos_t location;
} keytable[TABSIZE];
```

fgetpos Get the current file position
(continued)

```

struct record {
    char keyval[KEYLEN];
    char data[DATALEN];
};

int filesize;

/* Initialize keytable, which is a */
/* table of keys and locations      */
void bldtable(FILE *fileptr)
{
    struct record input;
    int index = 0;

    while (!feof(fileptr))
    {
        /* Store file pointer location */
        fgetpos(fileptr, &keytable[index].location);

        /* Read 1 record */
        fread(&input, sizeof(struct record), 1, fileptr);
        if (feof(fileptr) || ferror(fileptr))
            break;

        /* Save the keyval */
        memcpy(keytable[index].keyval, input.keyval, KEYLEN);
        index++;
    }

    filesize = index;
    return;
}

/* Find a match in the file to the key, */
/* and return complete record.          */
int findrec(FILE *fileptr, char keyval[KEYLEN],
            struct record *input)
{
    int index;

```


fgetpos Get the current file position
(continued)

```

        /* Search keytable for specified value */
        for (index = 0; index < filesize; ++index)
            if (memcmp(keyval, keytable[index].keyval, KEYLEN) == 0)
                break;

        if (index >= filesize)
            return (-1); /* Keyval not found */

        /* If found, read complete record from file */
        fsetpos (fileptr, &keytable[index].location);
        fread (input, sizeof(struct record), 1, fileptr);
        return (0);
    }

```

See Also `fseek`, `fsetpos`, `ftell`, `lseek`

fgets Get a string from a level 2 file

Synopsis `#include <stdio.h>`

```
p = fgets(buffer,length,fp);

char *p;          /* buffer pointer or NULL */
char *buffer;     /* buffer pointer */
int length;       /* buffer length in bytes */
FILE *fp;         /* file pointer */
```

Description The **fgets** function gets a string from the specified level 2 file, which must have been previously opened for input. Characters are copied from the file to the buffer until a new line (`\n`) has been copied, or `length-1` characters have been copied, or the end-of-file is hit. In any case, the buffer is terminated with a trailing null byte (`\0`).

Portability ANSI

Returns If the end-of-file is hit before any bytes are read, a **NULL** pointer is returned. If an I/O error occurs, a **NULL** pointer is returned and additional information is placed in the external integers `errno` and `_OSERR`. If no I/O error occurs and at least one byte was read from the file, the buffer argument is returned.

Example

```
/*
 * Assume that stdin contains the following lines:
 *
 * Hello, folks!
 * Goodbye, folks!
 */

#include <stdio.h>

void main(void)
{
    char *p,b[80];

    /* For the next two lines, p will point to b */
    p = fgets(b,sizeof(b),stdin);
    printf("b = %p, %sp = %p\n", b, b, p);
```

fgets Get a string from a level 2 file
(continued)

```

/* b contains "Hello, folks!" */
p = fgets(b,sizeof(b),stdin);
printf("b = %p, %sp = %p\n", b, b, p);

/* b now contains "Goodbye, folks!\n" */
p = fgets(b,sizeof(b),stdin);
printf("b = %p, %sp = %p\n", b, b, p);
}

```

See Also `errno`, `feof`, `ferror`, `fgetc`, `fopen`, `fread`, `getc`, `gets`, `_OSERR`

fileno Get the file number for a level 2 file

Synopsis `#include <stdio.h>`

```
fh = fileno(fp);

int fh;      /* file handle */
FILE *fp;    /* file pointer */
```

Description This function gets the file handle (the file number) associated with the specified file pointer. The file pointer must be one that was returned by the **fopen**, **freopen**, or **fdopen** function.

This function is implemented as a macro. Also, it does not check that the **fp** argument is a valid file pointer.

This function is not available if the **__STRICT_ANSI** flag has been defined.

Portability UNIX

Returns The return value is an integer representing the file handle.

See Also **fdopen**, **fopen**, **freopen**

findpath Locate a file in the current path

Synopsis `#include <dos.h>`

```
lock = findpath(filename);
```

```
BPTR lock;
const char *filename;
```

Description This function locates a file in the currently defined path. If the process is not a CLI process, it uses the path in effect when Workbench was loaded.

Portability AmigaDOS

Returns If the **findpath** function finds the file, it returns a lock on that directory even if it is the current directory. The lock must be unlocked with the AmigaDOS **UnLock** function. If the **findpath** function cannot find the file, it returns a `-1`. The value `NULL` is not used because `NULL` is a valid lock.

Example

```
#include <stdio.h>
#include <stdlib.h>
#include <dos.h>
#include <proto/dos.h>
#include <proto/exec.h>

void main(int argc, char *argv[])
{
    char path[256];
    long rc;
    BPTR lock;
    struct DosLibrary *DOSBase;

    rc = EXIT_FAILURE;

    if ((DOSBase = (struct DosLibrary *)
        OpenLibrary("dos.library", 0L)) == NULL)
    {
        printf("Couldn't open dos.library!\n");
    }
}
```

findpath Locate a file in the current path
(continued)

```

else
{
    if (argc < 2)
    {
        printf("You must enter a file to find!\n");
    }
    else
    {
        printf("Looking for \"%s\"... ",
            argv[1]);

        if ((lock = findpath(argv[1])) == ((BPTR)-1))
        {
            printf("Error!\n"
                "Cannot find \"%s\" in "
                "the current path\n", argv[1]);
        }
        else
        {
            printf("Found it!\n");
            if (getpath(lock, path) == 0)
            {
                printf("Path: \"%s\"\n",
                    path);
            }
            UnLock(lock);
            rc = EXIT_SUCCESS;
        }
    }
    CloseLibrary((struct Library *)DOSBase);
}
exit(rc);
}

```

See Also **getpath**, **UnLock** (*The AmigaDOS Manual, 3rd Edition*)

floor Get the floor of a real number

Synopsis `#include <math.h>`

```
x = floor(y);
```

```
double x,y;
```

Description This function returns the largest integer not greater than the specified real number.

Portability ANSI

Returns The result is a real number.

Example

```
#include <stdio.h>
#include <math.h>

void main(void)
{
    double r;

    r = floor(523.96);    /* r contains 523.0 */
    printf("floor(523.96) = %lf\n", r);
}
```

See Also `ceil`

flushall Flush all level 2 output buffers

Synopsis `#include <stdio.h>`

```
num = flushall(void);
```

```
int num; /* number of open files */
```

Description The `flushall` function flushes all level 2 output buffers and returns the number of level 2 files that are open. If an error occurs, the function continues to flush the remaining files and then returns a value of `-1`.

This function is not available if the `__STRICT_ANSI` flag has been defined.

Portability XENIX

Returns If an error occurs, the return value is `-1` (`EOF`). The appropriate error code is placed into the external integer `errno`, and additional information is placed in the external integer `_OSERR`.

See Also `errno`, `fclose`, `fcloseall`, `fflush`, `fopen`, `_OSERR`

fmod Floating-point modulus operations*Synopsis* `#include <math.h>``x = fmod(y,z);``double x,y,z;`*Description* The **fmod** function calculates the floating-point remainder of **y/z**. If the % (modulus) operation were defined for floating-point numbers, the expression would produce the following:`x = y % z;`*Portability* ANSI*Returns* This function returns the value **y** if the value **z** is 0. Otherwise, it returns a value that has the same sign as **y**, is less than **z**, and satisfies the following relationship:`y = (i * z) + x`The argument **i** is an integer.*Example*

```
#include <stdio.h>
#include <math.h>

void main(void)
{
    double r;

    r = fmod(5.7,1.5);      /* r contains 1.2 */
    printf("fmod(5.7, 1.5) = %lf\n", r);
}
```

See Also **modf**

fmode Change the mode of a level 2 file

Synopsis `#include <stdio.h>`

```
void fmode(fp,mode);

FILE *fp;    /* file pointer */
int mode;    /* 0 => mode A */
             /* 1 => mode B */
```

Description This function is used to change the translation mode of a level 2 file that has been opened with the **fopen**, **freopen**, or **fdopen** function.

In mode A, carriage returns are deleted on input, and a carriage return is inserted before each line feed on output. Mode A also detects the Control-Z character (0x1A) as a logical end-of-file mark. In mode B, all data are transferred with no changes.

For AmigaDOS, the default mode is B.

The file pointer is not checked for validity.

This function is not available if the `__STRICT_ANSI` flag has been defined.

Portability SAS/C

See Also **fdopen**, **fopen**, **freopen**

fopen Open a level 2 file

Synopsis `#include <stdio.h>`

```
fp = fopen(name, "mode");

FILE *fp;           /* file pointer */
const char *name;    /* file name */
const char *mode;    /* access mode string */
```

Description This function opens a file for buffered access. The name string can be any valid filename and may include a device code and directory path. The mode string specifies the values for each mode (Create, Trunc, Read, Write, Append, and Translate) with which you want to open the file.

Note: Do not place the `const` keyword on the declarations for the `name` and `mode` arguments in your program. For information about the `const` keyword, see the description of the Synopsis section at the beginning of this chapter.

The values of the modes Create, Trunc, Read, Write, Append, and Translate indicate how you want the file to be processed. The following table describes the effect of each setting of these modes.

Mode	Value	Effect
Create	yes	The file will be created if it does not already exist.
	no	The function will fail if the file does not already exist.
Trunc	yes	If the file exists, it will be truncated (marked as empty).
	no	If the file exists, its current contents will not be disturbed.
Read	yes	The file can be read with functions such as <code>fread</code> and <code>fgetc</code> . Also, the <code>fseek</code> function can be used to position the file before reading.
	no	The file cannot be read.

(continued)

fopen Open a level 2 file
(continued)

Mode	Value	Effect
Write	yes	The file can be written with functions such as fwrite and fputc . Also, the fseek function can be used to position the file before writing.
	no	The file cannot be written, but see Append below.
Append	yes	The file can be written, but it is automatically positioned to the current end-of-file before each write operation. This mode prevents existing data from being changed.
	no	Automatic positioning to the end-of-file is not done before a write operation. Also, writes are not allowed unless the Write mode value is Yes .
Translate	default	The external integer __fmode is used to set mode A or mode B as follows: <pre>if (__fmode & 0x8000) set mode B else set mode A</pre> For AmigaDOS, the external integer __fmode is normally 0x8000.
	mode A	On a read operation, each carriage-return character (\r) is deleted, and the Control-Z character is treated as a logical end-of-file mark. On a write operation, each line-feed character (\n) is expanded to a carriage return followed by a line feed.
	mode B	The data are unchanged as they are read or written.

fopen Open a level 2 file
(continued)

The following table shows the list of values specified by each mode string.

Mode String	Create	Trunc	Read	Write	Append	Translate
r	no	no	yes	no	no	default
w	yes	yes	no	yes	no	default
a	yes	no	no	no	yes	default
r+	no	no	yes	yes	no	default
w+	yes	yes	yes	yes	no	default
a+	yes	no	yes	no	yes	default
ra	no	no	yes	no	no	mode A
wa	yes	yes	no	yes	no	mode A
aa	yes	no	no	no	yes	mode A
ra+	no	no	yes	yes	no	mode A
wa+	yes	yes	yes	yes	no	mode A
aa+	yes	no	yes	no	yes	mode A
rb	no	no	yes	no	no	mode B
wb	yes	yes	no	yes	no	mode B
ab	yes	no	no	no	yes	mode B
rb+	no	no	yes	yes	no	mode B
wb+	yes	yes	yes	yes	no	mode B
ab+	yes	no	yes	no	yes	mode B

If the file is successfully opened, the function returns a pointer to a *buffered I/O control block*, which is defined in the header file `stdio.h`. Normally, you will not need to access any information in the control block directly, but you should be very careful not to disturb the block

fopen Open a level 2 file
(continued)

accidentally. A common C programming error is to accidentally mutilate one of these control blocks, which can cause garbage to be written into a file.

Portability ANSI

Mode A open modes are an extension to the ANSI standard, and you should not use them in programs that you want to be ANSI-compatible.

Returns A **NULL** pointer is returned if the file cannot be opened. Consult the external integers **errno** and **_OSERR** for detailed error information.

When a file is opened for both reading and writing, you should call the **fseek** or **rewind** function when switching from reading to writing or vice-versa. It is not necessary to do this when you begin writing after reading up to the end of the file.

Example

```
/* This is an example of using fopen to write a copy */
/* file routine. It returns 0 if the file copy was */
/* successful; otherwise, it returns a -1. */

#include <stdio.h>
copy (infile,outfile)
    char *infile,*outfile; /* input and output file names */
{
    FILE *in,*out;
    char buf[100];
    int i;

    /* open the input file */
    if ((in=fopen(infile,"r"))==NULL)
        return(-1);
    if ((out=fopen(outfile,"wt"))==NULL)
    {
        fclose(in);
        return(-1);
    }

    /* copy the file contents */
    while (i=fread(buf,1,100,in))
        if (fwrite(buf,1,i,out)!=i)
            break;
```

fopen Open a level 2 file
(continued)

```
        /* now set up the return */  
        i=(ferror(in)||ferror(out))?-1:0;  
  
        /* close the files */  
        fclose(in);  
        fclose(out);  
        return(i);  
    }
```

See Also `fclose`, `fdopen`, `fgetc`, `fgets`, `fputc`, `fputs`, `fread`, `freopen`,
`fwrite`, `open`

forkl, forkv Create a child process with an argument list or vector

Synopsis `#include <dos.h>`

```
error = forkl(prog,arg0,arg1,...,argn,NULL,env,procid);
error = forkv(prog,argv,env,procid);

cc = wait(procid);
complist = waitm(proclist);

int error;          /* error code */
char *prog;         /* program name */
char *arg0;         /* argument #0 */
char *arg1;         /* argument #1 */
char *argn;         /* argument #n */
char *argv[];       /* argument vector */

struct FORKENV *env; /* pointer to pseudo environment*/
                  /* structure (may be NULL)*/

int cc;
struct ProcID *procid; /* pointer to process ID structure*/
struct ProcID **proclist; /* address of pointer to linked*/
                        /* list of process IDs*/
struct ProcID *complist; /* pointer to linked list of*/
                        /* completed process IDs*/
```

Description The **forkl** and **forkv** functions create a *child process* by loading a new program as a concurrent process. The parent continues to execute until it calls either the **wait** or **waitm** function; that is, the parent and child are multiprogrammed. When the child process completes, the parent process (the *current program*) can get its completion code with the **wait** or **waitm** function.

To specify the arguments for the new program with the **forkl** function, specify a list of argument string pointers terminated by a **NULL** pointer (**arg0,arg1,...,argn,NULL**). To specify the arguments with the **forkv** function, specify a single pointer (**argv**) to an array of argument string pointers, with the array being terminated by a **NULL** pointer. Following UNIX conventions, the first argument (**arg0** or **argv[0]**) should be the program name and is normally the same as the **prog** argument. Under AmigaDOS, the arguments are all concatenated into a pseudo-command line, with a blank separating adjacent arguments and a carriage-return character at the end. The maximum size of this line is 255 bytes.

forkl, forkv Create a child process with an argument list or vector
(continued)

The pseudo-environment pointer **env**, if specified, contains optional data about default files and process execution. The **FORKENV** structure is defined in the file **dos.h** and has the following format:

```
struct FORKENV {
    long priority;      /* new process priority          */
    long stack;         /* stack size for new process      */
    BPTR std_in;        /* stdin file handle for new process */
    BPTR std_out;       /* stdout file handle for new process */
    BPTR console;       /* console window for new process   */
    struct MsgPort *msgport; /* msg port to receive          */
                                /* termination msg from child */
};
```

The child process is supplied with default values for any field left null. In addition, a **NULL** pointer may be passed for this parameter, which causes default values to be used for all items. If the default values are used, the child process is created with a priority of 0, a stack size of 8000 bytes, the current **stdin** and **stdout** files, and console for the parent process, and a new message port is created to receive the termination message.

The new process executes as a CLI-type task. This means it will be expecting file handles for **stdin** and **stdout** to be present, and a console task handler to exist. If the parent process is running as a Workbench process then it is possible for none of these to exist for the child process to inherit. If the parent process can be invoked from Workbench, extra effort should be made to ensure the presence of file handles the child may require. These file handles are BPTR values, as returned by the AmigaDOS **open** call.

The optional message-port field is provided to allow the parent process to detect when a child process has terminated while awaiting other events, such as Intuition menu events. The termination message format is similar to an Intuition message. The **TermMsg** structure is defined in the **dos/dos.h** file and has the following format:

```
struct TermMsg {
    struct Message msg; /* termination message from child */
    long class;         /* class == 0                      */
};
```

forkl, forkv Create a child process with an argument list or vector
(continued)

```

short type;                /* message type == 0          */
struct Process *process;    /* process ID of sending task */
long ret;                  /* return value              */
};

```

When a termination message is received, the parent process must still call the **wait** function to remove the message and unload the child process. If the message was removed from the port, it must be replaced by calling the **PutMsg** function before calling the **wait** function.

The **forkl** function loads the program from the current path using the AmigaDOS search procedure. If the calling program was loaded from the Workbench, the path used is the path at the time the Workbench was loaded. Alternatively, you can include a path specification as part of the program name.

Upon successful creation of the child process, the **forkl** function fills in the process ID structure. The address of this structure must be passed as the last argument. The process ID structure is defined in the **dos.h** file and has the following format:

```

struct ProcID {             /* packet returned from fork */
    struct ProcID *nextID;   /* link to next packet       */
    struct Process *process; /* process ID of child        */
    int UserPortFlag;
    struct MsgPort *parent;  /* termination msg destination */
    struct MsgPort *child;   /* child process' task msg port */
    BPTR seglist;           /* child process' segment list */
};

```

The **nextID** field may be used to link process ID structures into a simple linked list for use with the **waitm** function. The process field is the address of the process's task structure and is the value used with ROM Kernel functions, such as the **signal** function, that require a task ID parameter. The remaining fields are used by the **wait** function.

The **wait** or **waitm** function must be called for each child process to ensure that the child process terminates cleanly. The **wait** function takes as its single argument a pointer to the **ProcID** structure for the child task. It waits for a termination message from one particular process, replying to and discarding any other messages that arrive at the message port. The function returns the child process's completion code. This is the value that was passed to the **exit** function when the child terminated.

forkl, forkv Create a child process with an argument list or vector
(continued)

If any child processes share a message port, however, then the **waitm** function should be called to ensure that no termination messages are lost. The **waitm** function requires the address of a pointer to the first **ProcID** structure in a linked list of child process **ProcID** structures. It waits for one or more termination messages, removing the **ProcID** structure from the original linked list and inserting it into a linked list of terminated process **ProcID** structures. The completion code is placed in the **UserPortFlag** field of the structure. The function then returns a pointer to the first structure in the list of terminated process structures. Since the original list is updated, it may be reused to wait for the remaining child processes.

The **wait** or **waitm** function must be called for each child process before terminating the parent process. Otherwise, the child process will never be unloaded, and there is an excellent possibility that the system will crash.

Note: BCPL programs cannot be executed using the **forkl** or **forkv** function. These programs include all of the AmigaDOS routines under Workbench 1.3. The AmigaDOS routines from Workbench 2.0 will fork properly.

Portability SAS/C

Returns If the specified program file cannot be found, a -1 return is made, and additional error information can be found in the external integers **errno** and **_OSERR**. You must call the **wait** function to obtain the completion code from the child process.

Example The following program, **child.c**, creates a child process.

```
/*
 * This program creates a child process and displays the
 * return code. The child program name and arguments
 * are taken from the command line.
 */
#include <stdio.h>
#include <stdlib.h>
#include <dos.h>

struct ProcID child;
```

forkl, forkv Create a child process with an argument list or vector
(continued)

```
void main(int argc, char *argv[])
{
    int ret;
    if (argc < 2)
    {
        printf("no program specified\n");
        printf("usage: fork program [arg1] ... [argn]\n");
        exit(EXIT_FAILURE);
    }

    printf("parent: beginning fork of %s\n",argv[1]);
    if (forkv(argv[1],&argv[1],NULL,&child) == -1)
    {
        printf("error forking child\n");
        exit(EXIT_FAILURE);
    }
    else
    {
        ret = wait(&child);
    }
    printf("parent: %s finished, ret = %d\n",argv[1],ret);
}
```

The following program, `parent.c`, forks multiple child processes.

```
/*
 * This example forks multiple child processes
 */
#include <stdio.h>
#include <stdlib.h>
#include <dos.h>

char *child1_argv[] = {"task1",      /* program name */
                      "argument1",  /* 1st argument */
                      "argument2",  /* etc., etc.   */
                      NULL};
```

forkl, forkv Create a child process with an argument list or vector
(continued)

```

char *child2_argv[] = {"task2",      /* program name */
                      "argument1",   /* 1st argument */
                      "argument2",   /* etc., etc.   */
                      NULL};

char *child3_argv[] = {"task3",      /* program name */
                      "argument1",   /* 1st argument */
                      "argument2",   /* etc., etc.   */
                      NULL};

void main(void)
{
    struct ProcID *children, *terminated, *task;
    struct ProcID child1, child2, child3;
    int taskno;

    if (forkv(child1_argv[0], child1_argv, NULL, &child1) == -1)
    {
        printf("error forking child1\n");
        exit(EXIT_FAILURE);
    }

    if (forkv(child2_argv[0], child2_argv, NULL, &child2) == -1)
    {
        printf("error forking child2\n");
        exit(EXIT_FAILURE);
    }

    if (forkv(child3_argv[0], child3_argv, NULL, &child3) == -1)
    {
        printf("error forking child3\n");
        exit(EXIT_FAILURE);
    }

    child3.nextID = NULL;
    child2.nextID = &child3;
    child1.nextID = &child2;

    children = &child1;

```

forkl, forkv Create a child process with an argument list or vector
(continued)

```

while(children)      /* wait until no more children */
{
    /* must pass ADDRESS of pointer */
    terminated = waitm(&children);
    for (task = terminated; task != NULL; task = task->nextID)
    {
        if (task == &child1)
        {
            taskno = 1;
        }
        else if (task == &child2)
        {
            taskno = 2;
        }
        else if (task == &child3)
        {
            taskno = 3;
        }
        printf("task %d terminated, value = %d\n",
            taskno, task->UserPortFlag);
    }
}
}

```

See Also **exit, wait, waitm; LoadSeg, CreateProc, Execute, and Open** in *The AmigaDOS Manual, 3rd Edition*; **System** in *The AmigaDOS Manual, 3rd Edition*.

fprintf Formatted print

Synopsis `#include <stdio.h>`

```
length = fprintf(fp,fmt,arg1,arg2,...);

int length;           /* number of characters generated */
FILE *fp;             /* file pointer */
const char *fmt;      /* format string */
type *argn;           /* arguments */
```

Description This function produces an output stream of ASCII characters, and sends the output to the level 2 file specified by the `fp` argument.

The `fmt` argument points to a string that contains ordinary characters and conversion specifications that indicate how you want the arguments `arg1`, `arg2`, and so on to be printed. The ordinary characters are copied to the output, but the conversion specifications are replaced with the correctly formatted values of the arguments `arg1`, `arg2`, and so on. The first conversion specification is replaced with the formatted value of `arg1`, the second specification is replaced with the value of `arg2`, and so on. In some cases, as described below, a conversion specification may process more than one argument.

Each conversion specification must begin with a percent character (%). To place an ordinary percent into the output stream, precede it with another percent in the `fmt` string. That is, `%%` will send a single percent character to the output stream. A specification has the following format:

`%[flags][width][.precision][size]type`

The brackets ([]) indicate optional fields. Each field is defined as follows:

flags controls output justification and the printing of signs, blanks, decimal places, and hexadecimal prefixes.

If any flag characters are used, they must appear after the percent. Valid flags are as follows:

- (minus) causes the result to be left-justified within the field specified by *width* or within the default width.
- + (plus) causes a plus or minus sign to be placed before the result. This flag is used in conjunction with the various numeric conversion types. If it is absent, the sign character is generated only for a negative number.

fprintf Formatted print
(continued)

- blank causes a leading blank for a positive number and a minus sign for a negative number. This flag is similar to the plus. If both the plus and the blank flags are present, the plus takes precedence.
- # (pound) causes special formatting. With the **o**, **x**, and **X** types, the pound flag prefixes any nonzero output with **0**, **0x**, or **0X**, respectively. With the **f**, **e**, and **E** conversion types, the pound flag forces the result to contain a decimal point. With the **g** and **G** types, the pound flag forces the result to contain a decimal point and retain trailing zeroes.
- 0 (zero) pads the field width with leading zeros instead of spaces for the **d**, **i**, **o**, **u**, **x**, **X**, **e**, **E**, **f**, **g**, and **G** conversion types. If the minus flag is also used, the zero flag is ignored. If a **precision** is specified, the zero flag is ignored for conversion types **d**, **i**, **o**, **u**, **x**, and **X**. Behavior of the zero flag is undefined for the remaining conversion types.

width specifies the *field width*, which is the minimum number of characters to be generated for this format item.

The width is a nonnegative number that specifies the minimum field width. If fewer characters are generated by the conversion operation, the result is padded on the left or right (depending on the minus flag described above). A blank is used as the padding character unless *width* begins with a zero. In that case, zero padding is performed. If the minus flag appears, padding is performed with blanks. *width* specifies the minimum field width, and it will not cause lengthy output to be truncated. Use the precision specifier for that purpose.

If you do not want to specify the field width as a constant in the format string, you can code it as an asterisk (*), with or without a leading zero. The asterisk indicates that the width value is an integer in the argument list. See the examples for more information on this technique.

precision specifies the *field precision*, which is the required precision of numeric conversions or the maximum number of characters to be copied from a string, depending on the *type* field.

fprintf Formatted print
(continued)

The meaning of the precision item depends on the field type, as follows:

Type	Meaning
c	The precision item is ignored.
d, i, o, u, x, X	The precision is the minimum number of digits to appear. If fewer digits are generated, leading zeroes are supplied.
e, E, f	The precision is the number of digits to appear after the decimal point. If fewer digits are generated, trailing zeroes are supplied.
g, G	The precision is the maximum number of significant digits.
s	The precision is the maximum number of characters to be copied from the string.

As with the width item, you can use an asterisk for the precision to indicate that the value should be picked up from the next argument.

size can be either **L** for long double, **l** for large size, or **h** for small size. When used with the **d, i, o, u, x, or X** conversion specifiers, **h** and **l** select argument types of **short *** and **long ***, respectively. When used with the **e, E, f, g, or G** conversion specifiers, the **L** specifies a long double argument instead of a double.

type specifies the type of argument conversion to be done. Valid conversion types are as follows:

- c** specifies single-character conversion. The associated argument must be an integer. The single character in the right-most byte of the integer is copied to the output.
- d** specifies decimal-integer conversion. The associated argument must be an integer, and the result is a string of

fprintf Formatted print
(continued)

digits preceded by a sign. If the plus and blank flags are absent, the sign is produced only for a negative integer. If the large size modifier is present, the argument is taken as a long integer.

- e** specifies double-precision floating-point conversion. The associated argument must be a double-precision floating-point number, and the result has the form:

−d.dd $\mathbf{e}$ −ddd

d is a single decimal digit, *dd* is one or more digits, and *ddd* is an exponent of at least two digits. The first minus sign is omitted if the floating-point number is positive, and the second minus sign is omitted if the exponent is positive. The plus and blank flags dictate whether there will be a sign character emitted if the number is positive. The number of digits before the decimal point depends on the magnitude of the number, and the number after the decimal point depends on the requested precision. The value is rounded to the specified number of digits. If no precision is specified, the default is six decimal places.

- E** specifies double-precision floating-point conversion. This is exactly the same as type **e** except that the result has the form:

−d.dd $\mathbf{E}$ −ddd

- f** specifies double-precision floating-point conversion. The associated argument must be a double-precision floating-point number, and the result has the form:

−dd.dd

dd indicates one or more decimal digits. The minus sign is omitted if the number is positive, but a sign character will still be generated if the plus or blank flag is present. The number of digits before the decimal point depends on the magnitude of the number, and the number after the decimal point depends on the requested precision. If no precision is specified, the default is six decimal places. If the precision is specified as 0, or if there are no nonzero digits to the right of the decimal point, then the decimal point is omitted unless the pound (#) flag is specified.

fprintf Formatted print
(continued)

- g** specifies double-precision floating-point conversion (general form). The associated argument must be a double-precision floating-point number, and the result is in the **e** or **f** format, depending on which gives the most compact result. The **e** format is used only when the exponent is less than -4 or greater than the specified or default precision. Trailing zeroes are eliminated, and the decimal point appears only if any nonzero digits follow it.
- G** specifies double-precision floating-point conversion (general form). This is identical to the **g** format, except that the **E** type is used instead of the **e** type.
- i** specifies decimal-integer conversion. The associated argument must be an integer, and the result is a string of digits preceded by a sign. If the plus and blank flags are absent, the sign is produced only for a negative integer. If the large size modifier is present, the argument is taken as a long integer.
- n** specifies the argument will be a pointer to an integer into which is written the number of characters written so far by this call to the **fprintf** function. If the large size flag is on, the argument must be a pointer to a long integer. If the small size flag is on, the argument must be a pointer to a short integer.
- o** specifies octal-integer conversion. The associated argument is taken as an unsigned integer, and it is converted to a string of octal digits. If the large size modifier is present, the argument must be a long integer.
- p** specifies pointer conversion. The associated argument is taken as a data pointer, and it is converted to hexadecimal representation. Under AmigaDOS, the pointer is printed as 8 hexadecimal digits, with leading zeroes if necessary.
- P** specifies pointer conversion. This is the same as the **p** format, except that uppercase letters are used as hexadecimal digits. This conversion type is an extension to the ANSI standard. Do not use this extension if you want your program to be ANSI-compatible.

fprintf Formatted print
(continued)

- s** specifies string conversion. The associated argument must point to a null-terminated character string. The string is copied to the output, but the null byte is not copied.
- u** specifies unsigned decimal integer conversion. The associated argument is taken as an unsigned integer, and it is converted to a string of decimal digits. If the large size modifier is present, the argument must be a long integer.
- x** specifies hexadecimal-integer conversion. The associated argument is taken as an unsigned integer, and it is converted to a string of hexadecimal digits with lowercase letters. If the large size modifier is present, the argument is taken as a long integer.
- X** specifies hexadecimal-integer conversion. This is the same as the **x** format, except that uppercase letters are used as hexadecimal digits.

Portability ANSI

Returns This function returns the number of output characters generated. If an error occurs, the **fprintf** function returns a negative value and places additional information in the external integers **errno** and **_OSERR**.

Example

```

/*
 * This example prints a message indicating whether
 * the function argument is positive or negative.
 * In the second printf, the width and precision
 * are 15 and 8, respectively.
 */
#include <stdio.h>
#include <math.h>

void pneg(double value)
{
    char *sign;

    if (value < 0)
    {
        sign = "negative";
    }

```

fprintf Formatted print
(continued)

```
        else
        {
            sign = "not negative";
        }
        fprintf(stdout,"The number %E is %s.\n",value,sign);
        fprintf(stdout,"The number %*. *E is %s.\n",15,8,value,sign);
    }

void main(void)
{
    pneg(37.8);
    pneg(-18.2);
}
```

See Also `errno`, `fscanf`, `_OSERR`, `printf`, `scanf`, `sprintf`, `sscanf`

fputc Put a character to a level 2 file

Synopsis `#include <stdio.h>`

```
r = fputc(c,fp);

int r;      /* EOF or c */
int c;      /* character to be output */
FILE *fp;   /* level 2 file pointer */
```

Description This function puts a single character to the specified level 2 file.

Portability ANSI

Returns If successful, this function returns the character `c`; otherwise, it returns `EOF`. For disk files, an `EOF` return usually means that the disk is full. However, this type of return can also occur if the device is write-protected or if a write error occurs. In any case, additional error information can be found in the external integers `errno` and `_OSERR`.

See Also `errno`, `fopen`, `fputchar`, `_OSERR`, `putc`, `putchar`

fputc Put a character to `stdout`

Synopsis `#include <stdio.h>`

```
r = fputc(c);
```

```
int r;      /* EOF or c */
int c;      /* Character to be output */
```

Description This function puts a single character to `stdout`.

This function is not available if the `_STRICT_ANSI` flag has been defined.

Portability XENIX

Returns If successful, this function returns the character `c`; otherwise, it returns `EOF`. For disk files, an `EOF` return usually means that the disk is full. However, this type of return can also occur if the device is write-protected or if a write error occurs. In any case, additional error information can be found in the external integers `errno` and `_OSERR`.

See Also `errno`, `fopen`, `fputc`, `_OSERR`, `putc`, `putchar`

fputs Put a string to a level 2 file

Synopsis `#include <stdio.h>`

```
error = fputs(s,fp);

int error;          /* non-zero if error */
const char *s;      /* string pointer */
FILE *fp;           /* file pointer */
```

Description The **fputs** function writes the string **s** to a level 2 file that was previously opened for output. The string must be terminated by a null byte, which is not written.

Portability ANSI

Returns If an error occurs, the return value is **EOF**; otherwise, it is 0. Additional error information can be found in the external integers **errno** and **_OSERR**.

Example

```
/*
 * This example writes the following two lines to stdout:
 *
 * This is the first line
 * This is the second line
 */
#include <stdio.h>

void main(void)
{
    puts("This is the first line");
    fputs("This is ",stdout);
    puts("the second line");
}
```

See Also **errno**, **ferror**, **fopen**, **fputc**, **puts**

fqsort Sort an array of floating-point numbers

Synopsis `#include <stdlib.h>`

```
void fqsort(fa,n);
```

```
float *fa;    /* pointer to float array */
size_t n;    /* number of elements in array */
```

Description The **fqsort** function sorts the specified array of floating-point numbers using the ACM 271 algorithm, more popularly known as Quicksort. This function is not available if the `__STRICT_ANSI` flag has been defined.

Portability SAS/C

Returns This function returns a `void`.

See Also `dqsort`, `lqsort`, `qsort`, `sqsort`, `tqsort`

fread Read and write blocks

Synopsis `#include <stdio.h>`

```
a = fread(b, bsize, n, fp);

size_t a;      /* actual number of blocks */
void *b;       /* pointer to first block */
size_t bsize;  /* size of block in bytes */
size_t n;      /* maximum number of blocks */
FILE *fp;      /* file pointer */
```

Description This function uses level 2 I/O operations to read blocks of data. Each block contains **bsize** bytes and up to **n** blocks are stored into contiguous memory locations beginning at location **b**.

Blocks are read until **n** blocks have been stored or until the end-of-file is hit. If the end-of-file is hit in the middle of a block, that partial block will be stored in the **b** array, but it will not be included in the function return value. In other words, the return value indicates the number of complete blocks that were read.

Portability ANSI

Returns This function returns the number of complete blocks that were read.

See Also `fclose`, `feof`, `ferror`, `fgetc`, `fopen`, `fputc`, `fseek`, `fwrite`

free Free memory

Synopsis `#include <stdlib.h>`

```
void free(b);
```

```
void *b;    /* block pointer */
```

Description The **free** function releases a block of memory that was previously obtained using the **calloc**, **malloc**, or **realloc** function. For compatibility with some versions of UNIX, the block is not actually returned to the free space pool until the next time you call the **calloc**, **malloc**, **realloc**, or **free** function. Then, if that next call is to the **realloc** function and the block being reallocated is the one that was just freed, the **realloc** function will proceed correctly. In other words, you can ask the **realloc** function to reallocate a block that was freed as long as you have not called the **calloc**, **malloc**, or **realloc** function in the meantime.

Portability ANSI

Example

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

struct LIST
{
    struct LIST *next;
    char text[2];
};

void main(int argc, char *argv[])
{
    struct LIST *p;
    struct LIST *q;
    struct LIST list;
    char b[256];
    int x;
```

free Free memory
(continued)

```

printf("\nBegin new group...\n");
for (q = &list; ; q = p)
{
    printf("Enter a text string: ");
    if (fgets(b,sizeof(b),stdin) == NULL)
    {
        break;
    }
    if (b[0] == NULL)
    {
        if (q == &list)
        {
            printf("\n");
            exit(EXIT_SUCCESS);
        }
        break;
    }
    x = sizeof(struct LIST) - 2 + strlen(b) + 1;
    p = (struct LIST *)malloc(x);
    if (p == NULL)
    {
        printf("No more memory\n");
        exit(EXIT_FAILURE);
    }
    q->next = p;
    p->next = NULL;
    strcpy(p->text, b);
}

printf("\n\nTEXT LIST...\n");

```

free Free memory
(continued)

```

/*
 * You must be sure to copy the next pointer from
 * the current block before you free it.  Some
 * systems rely on a side-effect to be able to
 * access the memory after it is freed -- this is
 * BAD PROGRAMMING PRACTICE!
 */
p = list.next;
while(p != NULL)
{
    q = p->next;
    printf(p->text);
    free((char *)p);
    p = q;
}
list.next = NULL;
}

```

See Also `calloc`, `getmem`, `malloc`, `rbrk`, `realloc`, `rlsmem`, `sbrk`

freopen Reopen a level 2 file

Synopsis `#include <stdio.h>`

```
fpr = freopen(name, mode, fp);
```

```
FILE *fpr;           /* file pointer after re-opening */
const char *name;     /* file name */
const char *mode;     /* access mode */
FILE *fp;            /* current file pointer */
```

Description This function reopens a level 2 file. That is, it attaches a new file to a previously used file pointer. The previous file is automatically closed before the file pointer is reused. The **name** and **mode** arguments are the same as those for the **fopen** function.

Portability ANSI

Returns If successful, this function returns the file pointer.

Check the return code for the value **NULL**; the same errors as defined for the **fopen** function may occur. Also, for complete portability, do not assume that the **fpr** and **fp** pointers are identical. Use the **fpr** pointer to access the reopened file, not the **fp** pointer.

See Also **fdopen**, **fopen**

frexp Split floating-point value

Synopsis `#include <math.h>`

```
f = frexp(v, xp);

double f;    /* fraction */
double v;    /* value */
int *xp;     /* exponent pointer */
```

Description The **frexp** function splits the floating-point value **v** into its fraction (mantissa) and exponent parts.

Portability ANSI

Returns This function returns the mantissa as a double-precision floating-point number whose absolute value is greater than or equal to 0.5 and less than 1.0. The exponent is returned as an integer whose absolute value is less than 1024. If the value **v** is 0, both returned values will be 0.

See Also **fmod**, **ldexp**, **__matherr**, **modf**

fscanf Formatted input conversions

Synopsis `#include <stdio.h>`

```
n = fscanf(fp,fmt,arg1,arg2,...);

int n;           /* number of input items matched, or EOF */
FILE *fp;        /* file pointer (fscanf only) */
const char *fmt; /* format string */
type *argx;      /* pointers to input data areas */
```

Description This function performs formatted input conversions on text obtained from a specified level 2 file. The input characters are read and checked against the format string, which may contain any of the following:

white space

Any number of spaces, horizontal tabs, or new-line characters cause input to be read up to the next character that is not white space.

ordinary characters

Any character that is not white space and is not the percent sign (%) must match the next input character. Use a double percent (%%) in the format string to match a single percent in the input. If there is not an exact match, scanning stops, and the function returns.

conversion specification

This is a multicharacter sequence that indicates how the next input characters are to be converted. The following paragraphs describe this conversion specification.

The conversion specification follows this format:

```
%*n(1 | h)t
```

The various fields are defined as follows:

- % introduces a conversion specifier. If you want to match a percent sign in the input, indicate this by a double percent (%%) in the format string.
- * means that the conversion should be performed, but the result should not be stored. The asterisk is optional. You should not specify a pointer for any conversion specification that uses the asterisk (*) to suppress conversion.

fscanf Formatted input conversions
(continued)

- n* specifies the maximum input field width. This is an optional decimal number. This is used only with the *s* format.
- l* indicates that a long integer conversion should be performed. The letter *l* is optional. Note that the letters *l* and *h* are alternatives.
- h* indicates that a short conversion should be performed. The letter *h* is optional. If neither *l* nor *h* is specified, the default is an integer.
- t* stands for one of the following format characters: *c*, *d*, *e*, *f*, *g*, *i*, *n*, *o*, *s*, *u*, and *x*. These characters are described below.

If the conversion is successful and the assignment is not suppressed, the result is placed into the corresponding argument. The argument list must contain a pointer to an appropriate data item for each conversion specification that does not suppress assignment.

The function returns the number of conversion values that were assigned. This can be less than the number expected if the input characters do not agree with the format string. If an end-of-input is reached before any values are assigned, the return value is **EOF**.

The format characters listed previously for the *t* field specify how the input characters are to be converted. Leading white space is skipped in all cases except the *c* conversion.

- c* specifies character conversion. The corresponding argument must point to a character. The next input character is moved to that destination. No white space is skipped.
- d* specifies decimal number conversion. The corresponding argument must point to an integer or to a long integer. The latter applies if the *d* is preceded by an *l*. The input characters should be decimal digits, optionally preceded by a plus or minus sign.
- e,f,g* specifies floating-point conversion. These three types are identical. The corresponding argument must point to a floating-point number or a double-precision floating-point number. The latter applies if the type letter is preceded by an *l*.

The input characters must appear in the following sequence:

1. optional leading white space
2. an optional plus (+) or minus (−) sign
3. a sequence of decimal digits
4. an optional decimal point followed by 0 or more decimal digits

fscanf Formatted input conversions
(continued)

5. an optional exponent, consisting of the letter **e** or **E** followed by an optional plus or minus sign followed by one or more decimal digits

The input characters must follow this format, where brackets ([]) indicate an optional part:

[*whitespace*][*sign*]*digits*[.*digits*][*exponent*]

where

- n** indicates a character count. No input characters are read. The corresponding argument must point to an integer into which is written the number of input characters read so far.
- o** indicates an octal number. An octal number is expected, and the corresponding argument should point to an integer, or to a long integer if the **o** is preceded by an **l**.
- s** indicates a string. A character string is expected, and the corresponding argument should point to a character array large enough to hold the string and a terminating null byte. The input string is terminated by white space or the end-of-input. Also, if a maximum field width is specified, the output array size should be at least that width plus 1 because the reading of input characters will stop at the field width even if no white space has been hit.
- u** indicates an unsigned number. An unsigned decimal number is expected, and the corresponding argument should point to an unsigned integer, or to an unsigned long integer if the **u** is preceded by an **l**.
- x** indicates a hexadecimal number. A hexadecimal number is expected, and the corresponding argument should point to an integer, or to a long integer if the **x** is preceded by an **l**. The hexadecimal number can begin with the characters **0x** or **0X**, and case is not significant for the hexadecimal letters.

Portability ANSI

Returns The function returns the number of assignments that were made. For example, a return value of 3 indicates that conversion results were assigned to the arguments **arg1**, **arg2**, and **arg3**.

See Also **scanf**, **sscanf**

fseek Set a level 2 file position

Synopsis `#include <stdio.h>`

```
error = fseek(fp,rpos,mode);

int error;          /* non-zero if error */
FILE *fp;           /* file pointer returned from fopen() */
long int rpos;      /* relative file position */
int mode;           /* seek mode */
```

Description The **fseek** function moves the byte cursor of a level 2 file to a new position. The **mode** argument must be one of the following:

- SEEK_SET** seek from the beginning of the file. The **rpos** argument is the number of bytes from the beginning of the file. This value must be positive.
- SEEK_CUR** seek from the file's current position. The **rpos** argument is the number of bytes relative to the current position. This value can be positive or negative.
- SEEK_END** seek from the end of the file. The **rpos** argument is the number of bytes relative to the end of the file. This value must be negative or 0.

Portability ANSI

Returns A value of -1 is returned if an error occurs. The external integers **errno** and **_OSERR** contain additional error information.

See Also **errno**, **fopen**, **ftell**, **lseek**, **_OSERR**, **rewind**

fsetpos Reposition a file

Synopsis `#include <stdio.h>`

```
x = fsetpos(fp, pos);
```

```
int x;  
FILE *fp;  
const fpos_t *pos;
```

Description The **fsetpos** function positions the file pointed to by the **fp** argument to the position specified by the object pointed to by the **pos** argument. This object is of type **fpos_t**, which is defined in the **stdio.h** file.

The value of the object pointed to by the **pos** argument should be set by a previous call to the **fgetpos** function for the same file.

The **fsetpos** function can be used with most files, accessed either as text or binary. The **fsetpos** function clears the **EOF** indicator for the file on which it is called.

After a call to the **fsetpos** function on a stream that permits both reading and writing, the next file operation may be input or output.

Portability ANSI

Returns If successful, the **fsetpos** function returns 0. If it fails, the **fsetpos** function returns a nonzero value and stores an appropriate error code in the external integer **errno**. See the description of the external integer **errno** in Chapter 6 for the list of **errno** values.

Example See the example for the **fgetpos** function.

See Also **fgetpos**, **fseek**, **ftell**, **lseek**

fstat Get file status*Synopsis* #include <stat.h>

```
rc = fstat(file, st);

int rc;           /* return code */
int file;         /* file name */
struct stat *st;  /* stat info structure */
```

Description This function obtains information for the given file handle. Permission to read, write, or execute the file is not required.

This function is provided for compatibility with UNIX. For code that will be used only on the Amiga, use the AmigaDOS function **Examine** instead.

The information is placed into the **stat** structure pointed to by the **st** argument. The structure is defined in the file **stat.h** and contains information as follows:

```
struct stat {
    unsigned short  st_mode;           /* file mode */
    ino_t           st_ino;            /* inode number */
    dev_t           st_dev;            /* file system identifier */
    char            *st_rdev;          /* device identifier */
                                           /* (volume name) */
    short           st_nlink;          /* number of links */
    unsigned short  st_uid;            /* file owner user-id */
    unsigned short  st_gid;            /* file group user-id */
    off_t           st_size;           /* file size in bytes */
    time_t          st_atime;          /* time last accessed */
    time_t          st_mtime;          /* time last modified */
    time_t          st_ctime;          /* time last status change */
    short           st_type;            /* Amiga file type */
    char            *st_comment;       /* Amiga file comment */
};
```

fstat Get file status
(continued)

The following table lists defines that are combined with the logical OR operator to form the `st_mode` field. This list is found in the file `fcntl.h`.

Symbol	Meaning
<code>S_ISCRIPT</code>	The object has its script protection bit set.
<code>S_IPURE</code>	The object is an executable.
<code>S_IARCHIVE</code>	The file has its archive bit set.
<code>S_IREAD</code>	The file is readable.
<code>S_IWRITE</code>	The file is writable.
<code>S_IEXECUTE</code>	The file is executable.
<code>S_IDELETE</code>	The file is deletable.

Portability UNIX

Returns If the operation is successful, the function returns 0. Otherwise, it returns `-1` and places error information in the external integers `errno` and `_OSERR`.

See Also `chmod`, `errno`, `_OSERR`

ftell Get a level 2 file position

Synopsis `#include <stdio.h>`

```
apos = ftell(fp);

long int apos; /* absolute file position */
FILE *fp;      /* file pointer */
```

Description The **ftell** function returns a long integer value that is the current byte position in the file, relative to the beginning. It is equivalent to the following call:

```
apos = lseek(fp->_file, 0L, 1);
```

It is implemented as a function, not as a macro.

Portability ANSI

Returns For the **ftell** function, an error is indicated by a return value of `-1L`. The external integers **errno** and **_OSERR** contain additional error information.

See Also **errno**, **fopen**, **lseek**, **_OSERR**, **tell**

fwrite Write blocks to a level 2 file

Synopsis `#include <stdio.h>`

```
a = fwrite(b, bsize, n, fp);

size_t a;      /* actual number of blocks */
const void *b; /* pointer to first block */
size_t bsize;  /* size of block in bytes */
size_t n;      /* maximum number of blocks */
FILE *fp;      /* file pointer */
```

Description This function performs level 2 I/O operations to write blocks of data. Each block contains **bsize** bytes, and up to **n** blocks are stored into contiguous memory locations beginning at location **b**.

Blocks are written until **n** blocks have been sent or until the output device cannot accept any more. If the output device becomes full in the middle of a block, a partial block is written, but it is not included in the function return value. In other words, the return value indicates the number of complete blocks that were written.

Portability ANSI

Returns This function returns the number of complete blocks that were processed.

See Also `fclose`, `feof`, `ferror`, `fgetc`, `fopen`, `fputc`, `fread`, `fseek`

gcvt Convert a floating-point number to a string

Synopsis `#include <math.h>`

```
p = gcvt(v,dig,buffer);

char *p;          /* points to buffer */
double v;         /* floating point value */
int dig;          /* number of significant digits */
char *buffer;     /* output buffer */
```

Description This function converts the specified floating-point value into a null-terminated string in the output buffer. The string will be in one of two formats. First, the `gcvt` function attempts to produce `dig` significant digits in the FORTRAN F format. If that fails, it produces `dig` significant digits in the FORTRAN E format. Trailing zeroes are eliminated, if necessary.

Make sure that the specified buffer is large enough when using this function.

This function is not available if the `__STRICT_ANSI` flag has been defined.

Portability UNIX

Returns The function returns a pointer to the buffer.

Example

```
/*
 * This example displays 314150
 */
#include <stdio.h>
#include <stdlib.h>

void main(void)
{
    char s[100];

    printf("gcvt(3.1415e5,7,s) = %s\n",
          gcvt(3.1415e5,7,s));
}
```

See Also `ecvt`, `fcvt`

geta4 Establish addressability to the global data area

Synopsis `#include <dos.h>`

`void geta4(void);`

Description The **geta4** function establishes addressability to the global data area. The **geta4** function works as if you compiled your subroutine with the **saveds** option or put the **__saveds** keyword in the declaration. The **geta4** function is provided only for compatibility with other compilers. The **saveds** option and **__saveds** keyword are preferred over the **geta4** function.

Portability AmigaDOS

getasn Get assigned device

Synopsis `#include <dos.h>`

```
dlist = getasn(name);

struct DeviceList *dlist; /* pointer to device list entry */
                        /* for assignment */
const char *name;        /* assigned name */
```

Description Use the **getasn** function to search the device list for a name. If the name is found, the function returns a pointer to the device structure defined in the `dos/dosextens.h` file. You can use the **getasn** function to check whether a logical name is assigned. The **getasn** function can locate logical assigns made with the **assign** command. It can also find volume and device names.

The following example shows sample syntax for this function:

```
if (getasn("INCLUDE")) open ("INCLUDE:name")
```

Portability AmigaDOS

Returns The value **NULL** is returned if the value specified in the **name** argument cannot be located in the device list. **NULL** is defined in the `stddef.h` file. If the name is found, a pointer to the name's device node is returned. Some pointers in the device node are BCPL pointers, and strings are BCPL strings. See the file `dos/dosextens.h` for more information.

Example

```
#include <stdio.h>
#include <stdlib.h>
#include <dos.h>

void main(void)
{
    char *type;
    struct DeviceList *volume;

    volume=getasn("LC"); /* get LC: device node */
    if (volume==NULL)
    {
        fprintf(stderr,"LC: not assigned\n");
        exit(EXIT_FAILURE);
    }
}
```

getasn Get assigned device
(continued)

```

    }

    switch (volume->dl_Type)
    {
        case DLT_DEVICE:
            type = " device";
            break;
        case DLT_VOLUME:
            type = " volume";
            break;
        default:
            type = "n assignment";
            break;
    }
    printf("LC: is a%s\n", type);
}

```

See Also `getenv`, `putenv`

getc Get a character from a file

Synopsis `#include <stdio.h>`

```
c = getc(fp);
```

```
int c;          /* return character or code */
FILE *fp;       /* file pointer */
```

Description This function gets a single character from the specified level 2 file. The **getc** function is implemented as a macro to maximize execution speed.

Portability ANSI

Returns If successful, the next input character is returned. Otherwise, this function returns **EOF**, which is defined in the file **stdio.h**. In the event of an **EOF** return, error information can be found in the external integers **errno** and **_OSERR**. Most programmers treat any **EOF** return as an indication of end-of-file. However, if you want to distinguish errors from an end-of-file condition, you should reset the external integer **errno** before calling the function and then analyze its contents when you receive an **EOF** return.

See Also **errno**, **fgetc**, **fgetchar**, **fopen**, **getchar**, **_OSERR**

getcd Get the current directory

Synopsis `#include <dos.h>`

```
error = getcd(drive,path);

int error;    /* 0 if successful */
int drive;    /* drive code */
char *path;   /* points to path area (255 bytes recommended) */
```

Description This function gets the current directory path for the specified disk drive. The drive specification is retained in this implementation for compatibility with other implementations. However, you should always set the `drive` argument to 0 because only a value of 0 (indicating the current drive) is supported under AmigaDOS. Any other value is considered an error.

The path area must be large enough to contain the expected path. The returned string contains the entire path, including the volume name of the device.

Portability AmigaDOS

Returns If the operation is successful, the function returns 0. Otherwise, it returns `-1` and places error information in the external integers `errno` and `_OSERR`.

See Also `errno`, `getcwd`, `_OSERR`

getch Get a character from `stdin` immediately

Synopsis `#include <stdio.h>`

```
c = getch(void);
```

```
int c;      /* return character or code */
```

Description This function gets a single unbuffered character from `stdin`. If the input is a console device, it will be put into RAW mode until a character is typed, then switched back to the normal buffered mode.

This function is not available if the `__STRICT_ANSI` flag has been defined.

Portability SAS/C

Returns If successful, the next input character is returned. Otherwise, the function returns `EOF`, which is defined in the file `stdio.h`. In the event of an `EOF` return, error information can be found in the external integers `errno` and `_OSERR`. Most programmers treat any `EOF` return as an indication of end-of-file. However, if you want to distinguish errors from an end-of-file condition, you should reset the external integer `errno` before calling the function and then analyze its contents when you receive an `EOF` return.

Note: This function provides the functionality found in the `getchar` function in most other compilers on most other platforms. A carriage return is not required before the `getch` function returns a character.

See Also `errno`, `fgetc`, `fgetchar`, `fopen`, `getchar`, `_OSERR`

getchar Get a character from `stdin`

Synopsis `#include <stdio.h>`

```
c = getchar(void);
```

```
int c;      /* return character or code */
```

Description This function gets a single character from `stdin`.

Portability ANSI

Returns If successful, the next input character is returned. Otherwise, the function returns `EOF`, which is defined in the file `stdio.h`. In the event of an `EOF` return, error information can be found in the external integers `errno` and `_OSERR`. Most programmers treat any `EOF` return as an indication of end-of-file. However, if you want to distinguish errors from an end-of-file condition, you should reset the external integer `errno` before calling the function and then analyze its contents when you receive an `EOF` return.

When reading from an AmigaDOS console device, such as the CLI window, the `getchar` function will not return any characters until the user presses the Return key. To see keystrokes as they are typed, either use the `getch` function or put the console device into RAW mode.

See Also `errno`, `fgetc`, `fgetchar`, `fopen`, `getch`, `_OSERR`

getclk Get or change system clock

Synopsis `#include <dos.h>`

```
void getclk(clock);
```

```
unsigned char clock[8];
```

Description The **getclk** function obtains the current setting of the system clock and places it into an 8-byte array as follows:

Byte	Contents
<code>clock[0]</code>	day of the week (0 for Sunday)
<code>clock[1]</code>	year - 1980
<code>clock[2]</code>	month (1 to 12)
<code>clock[3]</code>	day (1 to 31)
<code>clock[4]</code>	hour (0 to 23)
<code>clock[5]</code>	minute (0 to 59)
<code>clock[6]</code>	second (0 to 59)
<code>clock[7]</code>	hundredths (0 to 99)

Portability AmigaDOS

See Also `chgclk`, `errno`, `_OSERR`

getcwd Get the current working directory

Synopsis `#include <dos.h>`

```
p = getcwd(b,size);

char *p;    /*Same as b if successful, else NULL*/
char *b;    /*Points to path buffer*/
int size;   /*Size of path buffer*/
```

Description This function obtains the path name for the current working directory. If the buffer pointer **b** is not **NULL**, then the path string is placed there if it will fit, and the return pointer **p** is the same as the pointer **b**. If the pointer **b** is **NULL**, then the **malloc** function is used to obtain a buffer of **size** bytes to hold the path string. In the latter case, you should use the **free** function to release the buffer when you are finished with it.

The **getcd** function is often more efficient since it does not use memory allocation.

Portability UNIX

Returns If the operation is successful, the function returns a pointer to the buffer. Otherwise, it returns a **NULL** pointer and places error information in the external integers **errno** and **_OSERR**. Also, a **NULL** pointer is returned if the path string will not fit in the buffer or if a buffer cannot be allocated. In either of those cases, the integer **errno** is unchanged, and the integer **_OSERR** is reset.

See Also **errno**, **getcd**, **_OSERR**

getdfs Get disk free space

Synopsis `#include <dos.h>`

```
error = getdfs(drive,info);

int error;                /*0 if successful*/
const char *drive;        /*drive or volume name*/
                           /*NULL => current drive*/
struct InfoData *info;    /*disk information*/
```

Description This function obtains information about the specified disk drive, including the amount of free space available. If a **NULL** pointer is passed as the drive name, information is obtained about the current drive. The **InfoData** structure is defined in the file `libraries/dos.h`.

```
struct InfoData {
    long    id_NumSoftErrors;    /* number of soft errors on disk */
    long    id_UnitNumber;      /* unit on which disk is mounted */
    long    id_DiskState;       /* see defines in libraries/dos.h */
    long    id_NumBlocks;       /* number of blocks on disk      */
    long    id_NumBlocksUsed;    /* number of blocks in use      */
    long    id_BytesPerBlock;
    long    id_DiskType;        /* disk type code                */
    BPTR    id_VolumeNode;
    long    id_InUse;           /* flag, 0 if not in use        */
};
```

Portability AmigaDOS

Returns A return value of 0 indicates success. If the drive code is invalid or no disk is mounted on that drive, then the return value is `-1`. No additional information is provided in the external integers `errno` and `_OSERR`.

getdfs Get disk free space
(continued)

Example

```
/*
 * Compute number of bytes available on current drive.
 */
#include <stdio.h>
#include <stdlib.h>
#include <dos.h>

void main(void)
{
    struct InfoData info;
    long size;

    if (getdfs(0,&info) != 0)
    {
        printf("getdfs failed\n");
        exit(EXIT_FAILURE);
    }
    size = (info.id_NumBlocks - info.id_NumBlocksUsed)
        * info.id_BytesPerBlock;
    printf("Current drive has %d bytes free.\n",
        size);
}
```

getenv Get environment variable

Synopsis `#include <stdlib.h>`

```
var = getenv(name);
```

```
char *var;           /*environment variable pointer or NULL */
const char *name;    /*environment variable name */
```

Description This function searches the environment strings for one that has the following form:

name=var

The *name* is the function argument. If such a string exists, the function returns a pointer to the *var* portion, which is null-terminated. Otherwise, a **NULL** pointer is returned.

Under AmigaDOS, environment variables are obtained by reading the first line out of the file **ENV:name**, where *name* is the name you specify.

The memory returned by the **getenv** function is obtained with the **malloc** function, so it remains through the duration of your program. However, the next call to the **getenv** function destroys the previous results from **getenv**. If you want to return the memory, you can call the **free** function.

Portability ANSI

Returns The value **NULL** is returned if *name* cannot be located in the current environment. **NULL** is defined in the **stddef.h** file.

getenv Get environment variable
(continued)

```
Example    #include <stdio.h>
           #include <stdlib.h>

           char *path;

           void main(void)
           {
               path = getenv("PATH"); /* get PATH variable */
               if (path == NULL)
               {
                   fprintf(stderr, "No PATH variable\n");
                   exit(EXIT_FAILURE);
               }
               printf("PATH environment variable contains:\n"
                     "%s\n", path);
           }
```

See Also `putenv`

getfa Get the file attribute

Synopsis `#include <dos.h>`

```
fa = getfa(name);
```

```
int fa;           /* 1, -1, or 0 */
const char *name; /* file name */
```

Description This function determines whether or not the specified file is a directory. The status is returned in the **fa** argument and contains the following information:

Status	Meaning
1	directory file
0	error
-1	normal file

Portability AmigaDOS

Returns If the operation is unsuccessful, the function returns 0 and places error information in the external integers **errno** and **_OSERR**.

See Also **errno**, **_OSERR**

getfnl Get a filename list

Synopsis `#include <dos.h>`

```
n = getfnl(fnp,fna,fnasize,attr);

int n;                /* number of matching file names */
const char *fnp;      /* file name pattern */
char *fna;            /* file name array */
size_t fnasize;       /* size of file name array */
int attr;             /* file attribute */
```

Description This function gets all the filenames that match the specified pattern and attribute, and it places them into the filename array. Each name is stored as a null-terminated string, and the filename array is terminated by a null string (a string consisting of only a null byte). If the filename pattern includes a path prefix, that prefix is placed in front of each matching filename.

The filename pattern has the following form:

drive:path/node.extension

The function first strips off the drive and directory path portions and restricts its search to that area of the file system. The node and extension parts can contain any valid filename characters, including the wildcard pattern matching characters. AmigaDOS wildcard characters are supported by default. Setting the external integer location **msflag** to a nonzero value causes MS-DOS wildcard characters * and ? to be supported instead. Some examples are

df0:#?.c

finds all files on drive df0: that have .c as their extension. A file named **abc.c** would thus be placed in the array as **df0:abc.c**. This assumes that the external integer **msflag** is set to 0.

:abc/def/q*.x?

finds all files in the directory **:abc/def** that begin with the letter q and have extensions consisting of the letter x and one other letter. For example, one such name would be **:abc/def/queen.x**. This assumes that the external integer **msflag** is set to 1.

XYZ*

finds all files in the current directory that begin with **XYZ**. One example is **XYZ**. This assumes that the external integer **msflag** is set to 1.

getfni Get a filename list
(continued)

AmigaDOS makes no distinction between uppercase and lowercase in any part of the filename.

The attribute is an integer defined as follows:

Attribute	Meaning
0	nondirectory files
1	all files including directories

This function is not available if the `__STRICT_ANSI` flag has been defined.

Portability SAS/C

Returns The function return value is the number of strings stored in the array, not including the terminating null string. A value of `-1` is returned if the filename pattern is invalid or if there is not enough room in the filename array. In the first case, the external integer `_OSERR` will contain further error information.

Example

```

/*
 * This program constructs an array of pointers to all normal
 * files in the current directory that have an extension of
 * .c. Then the array is sorted into ASCII order.
 */
#include <stdio.h>
#include <stdlib.h>

extern long _OSERR;

void main(void)
{
    char names[3000], *pointers[300];
    int count, i;

    count = getfni("#?.c", names, sizeof(names), 0);

```

getfnl Get a filename list
(continued)

```

if (count > 0)
{
    if (strbpl(pointers,300,names) != count)
    {
        fprintf(stderr,"Too many file names\n");
        exit(EXIT_FAILURE);
    }
    strsrst(pointers,count);
    printf("Names Found:\n");
    for (i=0; i<count; i++)
    {
        printf("%s\n", pointers[i]);
    }
}
else
{
    if (_OSERR)
    {
        poserr("FILES");
    }
    else
    {
        fprintf(stderr,"Too many files\n");
    }
    exit(EXIT_FAILURE);
}
}

```

See Also `dfind`, `dnext`, `_OSERR`, `strbpl`, `strsrst`

getft Get the file time

Synopsis `#include <dos.h>`

```
ft = getft(name);
```

```
long ft;           /* file time or -1 if error */
const char *name; /* file name */
```

Description This function gets the time and date information associated with the specified file. This information usually indicates when the file was created or last updated. This function returns the file time expressed as the number of seconds since 00:00:00 Greenwich Mean Time, January 1, 1970. The function `ctime` may be used to convert this value into date and time in an ASCII string.

Portability AmigaDOS

Returns If the `getft` function is successful, the file time (a long integer) is returned. Otherwise, a value of `-1` is returned. Additional error information can be found in the external integers `errno` and `_OSERR`.

See Also `ctime`, `errno`, `_OSERR`

getmem Get a memory block (short)

Synopsis `#include <stdlib.h>`

```
p = getmem(sbytes);
```

```
void *p;                /* block pointer */
unsigned int sbytes; /* number of bytes */
```

Description This function allocates a block from the memory pool and returns a pointer to the first byte in the block. If the pool does not currently contain a block of sufficient size, the **AllocMem** function is called to obtain more space from the operating system. If that step fails, a **NULL** pointer is returned.

This function is equivalent to the **malloc** function.

This function is not available if the **__STRICT_ANSI** flag has been defined.

Portability OLD

Returns A **NULL** pointer is returned if the block could not be allocated. Otherwise, a void pointer is returned, but it can be cast to any other pointer type.

Example See the example for the **malloc** function.

See Also **malloc**, **rlsmem**, **rlsml**, **sizmem**

getml Get a memory block (long)

Synopsis `#include <stdlib.h>`

```
p = getml(lbytes);
```

```
void *p;          /* block pointer */
long lbytes;      /* number of bytes */
```

Description This function allocates a block from the memory pool and returns a pointer to the first byte in the block. If the pool does not currently contain a block of sufficient size, the **AllocMem** function is called to obtain more space from the operating system. If that step fails, a **NULL** pointer is returned.

This function is equivalent to the **halloc** function.

This function is not available if the **__STRICT_ANSI** flag has been defined.

Portability OLD

Returns A **NULL** pointer is returned if the block could not be allocated. Otherwise, a character pointer is returned, but it can be cast to any other pointer type.

Example See the example for the **halloc** function.

See Also **getmem**, **halloc**, **rlsmem**, **rlsml**, **sizmem**

getpath Get the path for a specific directory/file

Synopsis

```
#include <dos.h>
error = getpath(lock, path);

int    error; /* 0 if ok, -1 if error */
BPTR   lock;  /* input */
char   *path; /* destination buffer */
```

Description This function returns the fully specified path string for the directory and/or file referenced by the lock. The **path** argument includes the volume name. The destination buffer must be large enough to contain the string.

Portability AmigaDOS

Returns If the **getpath** function is successful, it returns 0; otherwise, it returns -1.

See Also **findpath**, **Lock** and **UnLock** (*The AmigaDOS Manual, 3rd Edition*)

getreg Get 68000-specific registers

Synopsis `#include <dos.h>`

```
x = getreg(reg); /* obtain the current global static base */
```

```
long x;
int reg;
```

Description The built-in function **getreg** returns the current contents of a specified register. The valid register values (for the **reg** argument) are defined in the file **dos.h** as follows:

Value	Register Name
0	REG_D0
1	REG_D1
2	REG_D2
3	REG_D3
4	REG_D4
5	REG_D5
6	REG_D6
7	REG_D7

Value	Register Name
8	REG_A0
9	REG_A1
10	REG_A2
11	REG_A3
12	REG_A4
13	REG_A5
14	REG_A6
15	REG_A7

getreg Get 68000-specific registers
(continued)

Incorrect use of this function can cause serious problems. This function is used to read the processor registers directly. For example, you can use the **getreg** function to obtain the value of the system registers (for example, A4) to be passed to an interrupt chain.

Portability SAS/C

Returns The **getreg** function returns the current value of the register (a long integer).

See Also **emit**, **putreg**

gets Get a string from `stdin`

Synopsis `#include <stdio.h>`

```
p = gets(buffer);
```

```
char *p;          /*buffer pointer or NULL */
char *buffer;     /*buffer pointer */
```

Description The **gets** function copies characters from the `stdin` file (the standard input file) until a new line or end-of-file is reached. The new line, if present, is discarded and the buffer is terminated with a null byte (`\0`).

Make sure that your **gets** buffer can hold the largest line that will be encountered while reading the `stdin` file, because the function does not have any way to check for a maximum length.

Portability ANSI

Returns This function returns the buffer argument unless an I/O error occurs, in which case a `NULL` pointer is returned, and the external integer `errno` is set to describe the error.

Example

```
/*
 * Assume that stdin contains the following lines:
 * Hello, folks!
 * Goodbye, folks!
 */
#include <stdio.h>

void main(void)
{
    char *p,b[80];

    /* For the next two lines, p will point to b */
    p = gets(b); /* b contains "Hello, folks!" */
    printf("b = %p, %s\np = %p\n", b, b, p);

    p = fgets(b, sizeof(b), stdin); /* b now contains */
                                   /* "Goodbye, folks!\n" */
    printf("b = %p, %sp = %p\n", b, b, p);
```

gets Get a string from `stdin`
(continued)

```
p = gets(b);    /* Now, p is NULL */  
printf("b = %p, %sp = %p\n", b, b, p);  
}
```

See Also `errno`, `feof`, `ferror`, `fgetc`, `fgets`, `fopen`, `getc`

gmtime Unpack Greenwich Mean Time (GMT)

Synopsis `#include <time.h>`

```
ut = gmtime(t);
```

```
struct tm *ut;
const time_t *t;
```

Description This function unpacks a time value from the long integer form into a structure. Normally the time value represents the number of seconds since 00:00:00, January 1, 1970, Greenwich Mean Time. (The `time` function returns this type of number.)

This function expects a pointer as the argument. Do not pass the actual time value instead of the pointer. Also, the functions `gmtime` and `localtime` share a static data area for their return values, and a call to either one destroys the results of the previous call.

The structure `tm` is defined in the file `time.h` and has the following format:

```
struct tm {
    int tm_sec;      /* seconds          */
    int tm_min;      /* minutes         */
    int tm_hour;     /* hours since midnight */
    int tm_mday;     /* day of the month  */
    int tm_mon;      /* months since January */
    int tm_year;     /* years since 1900   */
    int tm_wday;     /* days since Sunday  */
    int tm_yday;     /* days since January 1 */
    int tm_isdst;    /* daylight savings time flag */
};
```

Portability ANSI

gmtime Unpack Greenwich Mean Time (GMT)
(continued)

Example

```
#include <stdio.h>
#include <time.h>

void main(void)
{
    struct tm *p;
    long t;

    time(&t);
    p = gmtime(&t);
    printf("GMT is %s\n", asctime(p));
}
```

See Also asctime, ctime, localtime, time

halloc Allocate a huge memory block

Synopsis `#include <stdlib.h>`

```
p = halloc(n);

void *p;          /* pointer to allocated memory    */
unsigned long n;  /* number of bytes to allocate */
```

Description This function allocates a huge block of memory. In the case of long integers, this function is identical to the `malloc` function. When using short integers, it allows allocation of more than 64 megabytes at a time, since the parameter is specified as an unsigned long integer.

Any memory allocated can be freed using the `free` function, or it will be automatically freed when the program terminates.

This function is not available if the `__STRICT_ANSI` flag has been defined.

Portability SAS/C

Returns The pointer is `NULL` if the request cannot be fulfilled.

Example

```
/*
 * Get space for 30,000 records, 30 bytes each
 */
#include <stdio.h>
#include <stdlib.h>

void main(void)
{
    char *p;

    p = halloc(90000L);
    if (p == NULL)
    {
        printf("cannot allocate record space\n");
        exit(EXIT_FAILURE);
    }
    free(p);
}
```

See Also `free`, `malloc`, `sbrk`

iabs Integer absolute value

Synopsis `#include <stdlib.h>`

```
ai = iabs(i);
```

```
int ai,i;
```

Description This function computes the absolute value of integers or short integers.
This function is not available if the `__STRICT_ANSI` flag has been defined.

Portability SAS/C

Returns This function returns an integer holding the absolute value of the parameter.

See Also **abs**

iomode Change the mode of a level 1 file

Synopsis `#include <fcntl.h>`

```
error = iomode(fh,mode);

int error;    /* error code          */
int fh;       /* file handle           */
int mode;     /* 0 => translated mode */
              /* 1 => raw mode         */
```

Description This function changes the mode of the specified level 1 file. When in translated mode, carriage returns are deleted on input, and a carriage return is inserted before each line feed on output. In RAW mode, all data in the file are transferred as is.

The **iomode** function affects only the software translation that is done by the level 1 I/O service functions.

Portability SAS/C

Returns A nonzero return value indicates that the specified file handle is not associated with a level 1 file.

See Also **open**

is..... Test character types

Synopsis `#include <ctype.h>`

```

t = isalnum(c); /* Test if alphanumeric character */
t = isalpha(c); /* Test if alphabetic character */
t = isascii(c); /* Test if ASCII character */
t = iscntrl(c); /* Test if control character */
t = iscsym(c); /* Test if C symbol character */
t = iscsymf(c); /* Test if C symbol lead character */
t = isdigit(c); /* Test if decimal digit character (0 to 9) */
t = isgraph(c); /* Test if graphic character */
t = islower(c); /* Test if lower case character */
t = isprint(c); /* Test if printable character */
t = ispunct(c); /* Test if punctuation character */
t = isspace(c); /* Test if space character */
t = isupper(c); /* Test if upper case character */
t = isxdigit(c); /* Test if hex digit character */
                  /* (0 to 9, A to F, a to f) */

int t; /* 0 if false, non-zero if true */
int c; /* character to test */

```

Description These functions test for various character types. If you include the file `ctype.h`, the functions are defined as macros. If you do not include the file `ctype.h`, these functions are resolved in the library. If you want to use the function versions but must include the file `ctype.h` for some other reason, use an `#undef` statement to undefine the appropriate character test macros.

You can use either characters or integers as arguments, but the macros are defined only over the integer range from -1 to 255. The functions, however, correctly handle the entire integer range. The reason -1 is included as a valid argument is to avoid a nonsensical result if you feed the `EOF` value to one of the macros or functions. `EOF` can be returned by the `getchar` function and other I/O functions, and if you pass it to any of the character test functions, the return value will be 0.

When you include the `ctype.h` file, these functions generate inline code to test the static array named `__ctype`. This array contains a bit mask for each of the 256 possible character values and for the integer value -1 . If you do not include the `ctype.h` file, you reduce your program size slightly at the expense of execution speed.

The `isascii`, `iscsym`, and `iscsymf` functions are not available if the `_STRICT_ANSI` flag has been defined.

is..... Test character types
(continued)

Portability SAS/C `isascii, iscsym, iscsymf`
ANSI all others

Returns These functions return a nonzero value if the test is true and 0 if the test is false.

Example

```
#include <stdio.h>
#include <ctype.h>

void main(void)
{
    int c;

    while((c = getchar()) != EOF)
    {
        printf("\n%c %s alphabetic.\n",
            c, isalpha(c) ? "is" : "is not");
    }
}
```

See Also `__ctype`

isatty Test a file descriptor for a terminal device

Synopsis `#include <fcntl.h>`

```
rc = isatty(fd)
```

```
int fd;          /* level 1 file descriptor */  
int rc;          /* return code */
```

Description This function takes a file descriptor as returned from a call to the level 1 file I/O function **open** and tests to see if the file descriptor is associated with a terminal device (such as a console window).

Portability UNIX

Returns If the file descriptor is associated with a terminal device, the routine returns 1. Otherwise, it returns 0.

See Also **open**

jrand48 Generate a random long integer (external seed)

Synopsis `#include <math.h>`

```
z = jrand48(seed);
```

```
long z; /* random long */
unsigned short seed[3]; /* seed value (high bits in seed[0]) */
```

Description This function generates random numbers using the linear congruential algorithm and 48-bit arithmetic. The `jrand48` function is provided for cases where several seeds are in use at the same time, so you can specify the seed on each function call.

This function is not available if the `__STRICT_ANSI` flag has been defined.

Portability UNIX

Returns The `jrand48` function returns signed long integers uniformly distributed over the interval from -2^{31} to $2^{31}-1$.

See Also `lcong48`, `mrnd`, `rand`, `srnd`, `srnd48`

labs Long integer absolute value

Synopsis `#include <stdlib.h>`

```
al = labs(l);
```

```
long int al,l;
```

Description This macro computes the absolute value of a long integer. It generates inline code to perform the conversion. The definition is

```
#define labs(i) ((i)<0?- (i):(i))
```

Portability ANSI

Returns This function returns a long integer holding the absolute value of the parameter.

See Also **abs, fabs, labs**

lcong48 Set linear congruence parameters

Synopsis `#include <math.h>`

```
void lcong48(parm);
```

```
unsigned short parm[7]; /* parameters */
```

Description The **lcong48** function allows an intricate initialization of the linear congruential algorithm. The algorithm is of the form:

$$X_{x+1} = (a * X_n + c) \bmod m$$

the value of m is 2^{**48} , and the default values for a and c are 0x5DEECE66D and 0xB, respectively. The array passed to the **lcong48** function is structured as follows:

Parameter	Value
<code>parm[0]</code>	bits 47-32 of value X_n
<code>parm[1]</code>	bits 31-16 of value X_n
<code>parm[2]</code>	bits 15-00 of value X_n
<code>parm[3]</code>	bits 47-32 of value a
<code>parm[4]</code>	bits 31-16 of value a
<code>parm[5]</code>	bits 15-00 of value a
<code>parm[6]</code>	value c

Whenever the **seed48** function is called, the **a** and **c** arguments are reset to their default values.

This function is not available if the **__STRICT_ANSI** flag has been defined.

Portability UNIX

See Also **jrand48**, **rand**, **seed48**, **srand**, **srand48**

ldexp Combine floating-point value

Synopsis `#include <math.h>`

```
v = ldexp(d,x);

double v;      /* value */
double d;      /* fraction */
int x;         /* exponent */
```

Description The **ldexp** function adds the integer **x** to the exponent in the argument **d**, which is the same as computing:

$$v = d * (2 ** x)$$

If **d** and **x** are the results of the **frexp** function, then the **ldexp** function performs the reverse of the **frexp** function. Also, if the absolute value of the resulting exponent is greater than 1,023, then the **__matherr** function is called with an overflow or underflow error indication.

Portability ANSI

Returns This function returns a double-precision floating-point number holding the result of the above equation.

See Also **fmod**, **frexp**, **__matherr**, **modf**

ldiv Return the long integer quotient and the remainder

Synopsis `#include <stdlib.h>`

```
res = ldiv(number, denom);

ldiv_t res;          /* resulting quotient and remainder */
long int number;     /* numerator for the divide      */
long int denom;      /* denominator for the divide      */
```

Description This function returns the quotient and the remainder obtained from performing a divide on long integers.

Portability ANSI

Returns This function returns a structure containing both the quotient and the remainder. The structure is defined in the `stdlib.h` file as follows:

```
typedef struct {
    long int quot;
    long int rem;
} ldiv_t;
```

Example

```
/*
 * Obtain both quotient and remainder
 * for a value divided by 10
 */
#include <stdio.h>
#include <stdlib.h>

void quotrem(long val)
{
    ldiv_t result;

    result = ldiv(val, 10L);

    printf("Quotient  = %ld\n", result.quot);
    printf("Remainder = %ld\n", result.rem);
}
```

ldiv Return the long integer quotient and the remainder
(continued)

```
void main(void)
{
    quotrem(42);
}
```

See Also **div**

localeconv Return information on locale formatting conventions
(continued)

```

char p_sep_by_space;    /* 1=space between currency symbol*/
                        /* and non-negative monetary */
                        /* quantity */
                        /* 0=no space */
char n_cs_precedes;    /* 1=currency symbol precedes */
                        /* negative monetary quantity */
                        /* 0=symbol succeeds quantity */
char n_sep_by_space;    /* 1=space between currency symbol*/
                        /* and negative monetary quantity */
                        /* 0=no space */
char p_sign_posn;       /* position of sign for positive */
                        /* monetary quantities */
char n_sign_posn;       /* position of sign for negative */
                        /* monetary quantities */
};

```

The decimal point character used to format nonmonetary values defaults to a period (.). The character used to separate groups of digits before the decimal point character in formatted nonmonetary values defaults to a comma (,).

Portability ANSI

Returns This function returns a pointer to the `lconv` structure for the current locale.

See Also `setlocale`

localtime Unpack local time*Synopsis* `#include <time.h>``ut = localtime(t);``struct tm *ut;
const time_t *t;`

Description This function unpacks a time value from the long integer form into a structure. Normally, the time value represents the number of seconds since 00:00:00, January 1, 1970, Greenwich Mean Time. (The `time` function returns this kind of number.) The `localtime` function adjusts the number for the local time zone.

This function expects a pointer as the argument. A common error is to pass the actual time value instead of the pointer. Also, the functions `gmtime` and `localtime` share a static data area for their return values, and a call to either one destroys the results of the previous call.

The structure `tm` is defined in the file `time.h`. See the description of the `gmtime` function for the format of the `tm` structure.

Portability ANSI

Example

```
#include <stdio.h>
#include <time.h>

void main(void)
{
    struct tm *p;
    long t;

    time(&t);
    p = localtime(&t);
    printf("Local time is %s\n",asctime(p));
}
```

See Also `asctime`, `ctime`, `gmtime`, `time`

log Natural logarithm function

Synopsis `#include <math.h>`

```
    r = log(x);
```

```
    double r, x;
```

Description The `log` function calculates the base E logarithm. This function requires a positive argument. If you enter a negative argument, the `__matherr` function is called with a **DOMAIN** error.

Portability ANSI

Returns This function returns a double-precision floating-point number that contains the base E logarithm of the parameter.

See Also `log10`, `__matherr`

log10 Base 10 logarithm function

Synopsis `#include <math.h>`

```
r = log10(x);
```

```
double r, x;
```

Description The `log10` function calculates the base 10 logarithm. This function requires a positive argument. If you enter a negative argument, the `__matherr` function is called with a **DOMAIN** error.

Portability ANSI

Returns This function returns a double-precision floating-point number that is the base 10 logarithm of the parameter.

See Also `log`, `__matherr`

longjmp Perform a long jump

Synopsis `#include <setjmp.h>`

```
void longjmp(save,value);

jmp_buf save; /* address of save area */
int value;    /* return value */
```

Description The `setjmp` function saves the current stack mark in a specified save area and returns a code of 0. A subsequent call to the `longjmp` function with the same save area will then cause control to return to the next statement after the original `setjmp` call, with `value` as the return code. If the return code is 0, it is forced to 1 by the `longjmp` function.

This mechanism is useful for quickly popping back up through multiple layers of function calls under exceptional circumstances.

Do not call the `longjmp` function with an invalid save area. It may disrupt the system. Do not use the `longjmp` function after the function calling the `setjmp` function has returned to its caller because the stack frame for that function no longer exists.

Portability ANSI

See Also `exit`, `setjmp`

lqsort Sort an array of long integers

Synopsis `#include <stdlib.h>`

```
void lqsort(la,n);
```

```
long *la;    /* pointer to long int array */
size_t n;    /* number of elements in array */
```

Description The **lqsort** function sorts the specified array of long integers using the ACM 271 algorithm, more popularly known as Quicksort.

This function is not available if the `__STRICT_ANSI` flag has been defined.

Portability SAS/C

See Also `dqsort`, `fqsort`, `qsort`, `sqsort`, `tqsort`

lrand48 Generate a random positive long integer (internal seed)

Synopsis `#include <math.h>`

```
y = lrand48(void);
```

```
long y;          /* random positive long */
```

Description This function generates random numbers using the linear congruential algorithm and 48-bit arithmetic. The `lrand48` function uses an internal 48-bit storage area for the seed value.

This function is not available if the `__STRICT_ANSI` flag has been defined.

Portability UNIX

Returns The `lrand48` function returns nonnegative long integers uniformly distributed over the interval from -2^{31} to $2^{31}-1$.

See Also `nrnd`, `rand`, `srnd`, `srnd48`

lsbrk Allocate memory

Synopsis `#include <stdlib.h>`

```
p = lsbrk(lbytes);
```

```
void *p;          /* block pointer */
long lbytes;      /* number of bytes */
```

Description The **lsbrk** function requests **lbytes** of memory from the system, adding the allocated block to a linked list of memory blocks to be returned to the system when the program terminates.

This function is provided for compatibility with previous versions of the compiler.

This function is not available if the **__STRICT_ANSI** flag has been defined.

Portability OLD

Returns An error is indicated by a **NULL** pointer. The **lsbrk** function returns the address of the block just allocated.

See Also **getmem, malloc, rbrk, sbrk**

lseek Set a level 1 file position

Synopsis `#include <fcntl.h>`

```
apos = lseek(fh,rpos,mode);

long apos; /* absolute file position */
int fh;    /* file handle */
long rpos; /* relative file position */
int mode;  /* seek mode */
```

Description This function moves the byte cursor of a level 1 file to a new position. The `mode` argument must be one of the following:

- 0 seek from the beginning of the file. The `rpos` argument is the number of bytes from the beginning of the file. This value must be positive.
- 1 seek from the current position of the file. The `rpos` argument is the number of bytes relative to the current position. This value can be positive or negative.
- 2 seek from the end of the file. The `rpos` argument is the number of bytes relative to the end of the file. This value must be negative or 0.

If the `lseek` function is asked to move 0 bytes relative to the current position, it simply returns the current file position.

Returns This function returns `-1` if an error occurs, in which case the external integers `errno` and `_OSERR` contain additional error information.

Portability UNIX

Example

```
/**
 * This program totals the number of bytes used by
 * all normal files in the current directory.
 */

#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h> /* for Level 1 I/O */

char names[8192]; /* holds file names */
```

lseek Set a level 1 file position
(continued)

```
void main(void)
{
    char *p;
    int f,n;
    long x,y;

    if (getfnl("#?",names,sizeof(names),0) <= 0)
    {
        printf("Can't build file name list\n");
        exit(EXIT_FAILURE);
    }
    for (x=0, n=0, p=names; *p!='\0'; p+=strlen(p)+1)
    {
        f = open(p,O_RDONLY);
        if (f < 0)
        {
            printf("Can't open \"%s\"\n",p);
            exit(EXIT_FAILURE);
        }
        y = lseek(f,0L,2);
        if (y < 0)
        {
            printf("Seek failure on \"%s\"\n",p);
            exit(EXIT_FAILURE);
        }
        x += y;
        n++;
        close(f);
    }
    printf("%d files, %ld bytes used\n",n,x);
}
```

See Also `errno`, `open`, `_OSERR`, `tell`

lstat Get file status

Synopsis `#include <stat.h>`

```
rc = lstat(file, st);

int rc;           /* return code */
const char *file; /* file name */
struct stat *st;  /* stat info structure */
```

Description This function obtains information for the given file. If the file is not in the current directory, the file path must be included as part of the filename. Permission to read, write, or execute the file is not required.

If the file referred to is a link, this function returns information on the link instead of the file to which it is linked.

This function works under all revisions of the operating system and is provided for compatibility with UNIX. For code that will be used only on the Amiga, use the AmigaDOS function **Examine** instead.

The information is placed into the **stat** structure pointed to by the **st** argument. The structure is defined in the file **stat.h** and contains information as follows:

```
struct stat {
    unsigned short  st_mode;      /* file mode */
    ino_t           st_ino;       /* inode number */
    dev_t           st_dev;       /* file system identifier */
    char            *st_rdev;     /* device identifier */
                                /* (volume name) */
    short           st_nlink;     /* number of links */
    unsigned short  st_uid;       /* file owner user-id */
    unsigned short  st_gid;       /* file group user-id */
    off_t           st_size;      /* file size in bytes */
    time_t          st_atime;     /* time last accessed */
    time_t          st_mtime;     /* time last modified */
    time_t          st_ctime;     /* time last status change */
    short           st_type;      /* Amiga file type */
    char            *st_comment;  /* Amiga file comment */
};
```

lstat Get file status
(continued)

The following table lists defines that are combined with the logical OR operator to form the `st_mode` field. This list is found in the file `fnctl.h`.

Symbol	Meaning
<code>S_ISCRIPT</code>	The object has its script protection bit set.
<code>S_IPURE</code>	The object is an executable.
<code>S_IARCHIVE</code>	The file has its archive bit set.
<code>S_IREAD</code>	The file is readable.
<code>S_IWRITE</code>	The file is writable.
<code>S_IEXECUTE</code>	The file is executable.
<code>S_IDELETE</code>	The file is deletable.

Portability UNIX

Returns If the operation is successful, the function returns 0. Otherwise, it returns `-1` and places error information in the external integers `errno` and `_OSERR`.

See Also `chmod`, `errno`, `_OSERR`

__main Standard preprocessing for the main module

Synopsis `#include <stdlib.h>`

```
void __stdargs __main(line);
```

```
char *line; /* ptr to command line that caused execution */
```

Description The `__main` function performs the standard preprocessing for the main module of a C program. It accepts a command line of the following form:

program-name arg1 arg2

It builds a list of pointers to each argument and the first pointer is to the program name. The `__main` function also opens the standard I/O files `stdin`, `stdout`, and `stderr`. `__main` calls the function `main` with the standard `argc` and `argv` parameters.

Unlike previous editions of the compiler, this function is declared with the `__stdargs` keyword.

The source code for this function is in the file `_main.c` in the `sc:source` directory.

For more information on `__main`, refer to Chapter 10, “Using Startup Modules, Autoinitialization, and Autotermination Functions,” in *SAS/C Development System User’s Guide, Volume 1: Introduction, Editor, Compiler*.

Portability SAS/C

See Also `main`, `__tinymain`

main Your main or principal function

Synopsis `#include "workbench/startup.h"`

```
int main(argc,argv);

int argc;          /* argument count */
union {
    char *args[];
    struct WBStartup *msg;
} argv;           /* argument vector */
```

Description This function does not actually exist in the library; you must supply one of these main programs in each of your applications. If you trace through the two startup modules `c.a` and `_main.c`, you will find that the module `c.a` passes control to the module `_main.c`, which then calls the function named `main`. Since the source code for both of these modules is supplied, you are free to change this initialization procedure for special applications. The standard version simulates UNIX's interface with C programs by setting up a vector, which is simply an array of pointers.

The `argv.args` array contains pointers to the command-line arguments, and the argument `argc` indicates how many pointers are in the array. For example, you can start the program `myprog` with the following command:

```
myprog abc def "ghi jkl"
```

Then, set up the `argv.args` array as follows:

```
argv.args[0] => "myprog"
argv.args[1] => "abc"
argv.args[2] => "def"
argv.args[3] => "ghi jkl"
```

The `argc` argument contains the value 4.

Under Workbench, there is no command line. In this case, the argument `argc` is 0 indicating no command or arguments, and the argument `argv.msg` is a pointer to the Workbench startup message structure.

Portability ANSI

main Your main or principal function
(continued)

Returns When the `main` function returns to its caller (normally the `_main.c` function), the program exits to AmigaDOS with a termination code of 0. If you want to pass a nonzero termination code back to AmigaDOS, use the `exit` or `__exit` function.

Example

```

/**
 * This program is intended to run only under CLI
 * and displays the command and any arguments
 */

int main(int argc, char *argv[])
{
    int i;

    printf("command = %s\n",argv[0]);
    for (i = 0; argc > 1; i++, argc--)
        printf("argument %d = %s\n",i,argv[i]);
    return(0);
}

/**
 * This program is intended to run only under WorkBench and
 * gets its arguments from the WorkBench message structure
 */

#include <workbench/startup.h>

int main(int argc, char *argv[])
{
    struct WBStartup *wbs;
    int i;

    if (argc != 0)
        exit(EXIT_FAILURE);

    wbs = (struct WBStartup)argv;

```


main Your main or principal function
(continued)

```

        printf("command = %s\n", wbs->sm_ArgList[0].wa_Name);
        for (i = 1; i < wbs->sm_NumArgs; i++)
            printf("argument %d = %s\n",
                i, wbs->sm_ArgList[i].wa_Name);
        return(0);
    }

/* This program is intended to run under either */
/* WorkBench or CLI.                            */

#include <stdio.h>
#include <workbench/startup.h>

int main(int argc, union {
                                char **args;
                                struct WBStartup *msg;
                            } argv)
{
    int i;

    if (argc != 0)
    {
        printf("command = %s\n",argv.args[0]);
        for (i = 0; argc > 0; i++, argc--)
            printf("argument %d = %s\n",
                i,argv.args[i]);
    }
    else
    {
        printf("command = %s\n",
            argv.msg->sm_ArgList[0].wa_Name);
        for (i = 1; i < argv.msg->sm_NumArgs; i++)
            printf("argument %d = %s\n",
                i,argv.msg->sm_ArgList[i].wa_Name);
    }
    return(0);
}

```

main Your main or principal function
(continued)

See Also **exit, __exit, __main**

malloc Allocate memory

Synopsis `#include <stdlib.h>`

```
b = malloc(n);

void *b;      /*block pointer */
size_t n;     /*number of bytes */
```

Description The **malloc** function allocates a block that is *n* bytes long and is aligned in such a way that you can cast the block pointer to any pointer type. If the block cannot be allocated, a **NULL** pointer is returned.

The **malloc** function can only allocate 64 megabytes at a time if short integers are used.

Portability ANSI

Returns A **NULL** pointer is returned if there is not enough space for the requested block.

Example

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

struct LIST
{
    struct LIST *next;
    char text[2];
};

void main(int argc, char *argv[])
{
    struct LIST *p;
    struct LIST *q;
    struct LIST list;
    char b[256];
    int x;

    printf("\nBegin new group...\n");
    for (q = &list; ; q = p)
    {
        printf("Enter a text string: ");
        if (fgets(b,sizeof(b),stdin) == NULL)
```

malloc Allocate memory
(continued)

```

    {
        break;
    }
    if (b[0] == NULL)
    {
        if (q == &list)
        {
            printf("\n");
            exit(EXIT_SUCCESS);
        }
        break;
    }
    x = sizeof(struct LIST) - 2 + strlen(b) + 1;
    p = malloc(x);
    if (p == NULL)
    {
        printf("No more memory\n");
        exit(EXIT_FAILURE);
    }
    q->next = p;
    p->next = NULL;
    strcpy(p->text, b);
}
printf("\n\nTEXT LIST...\n");
p = list.next;
while(p != NULL)
{
    q = p->next;
    printf("%s", p->text);
    free(p);
    p = q;
}
list.next = NULL;
}

```

See Also `calloc`, `free`, `getmem`, `rbrk`, `realloc`, `rlsmem`, `sbrk`

__matherr Math error handler

Synopsis `#include <math.h>`

```
a = __matherr(x);
```

```
int a;                                /* action code      */
struct __exception *x;                /* exception vector */
```

Description The `__matherr` function is called whenever one of the higher-level math functions detects an error. The exception vector structure is defined in the file `math.h` and contains information about the error as follows:

```
struct __exception
{
    int type;                        /* error type      */
    char *name;                      /* math function name */
    double arg1, arg2;               /* function arguments */
    double retval;                   /* proposed return value */
};
```

The codes for the `type` field in `struct __exception` are in the file `math.h`.

The standard library version of the `__matherr` function translates the error type into a UNIX error code that is placed into the external integer `errno`. Then the function returns an action code of 0 to indicate that the math function should simply use the proposed return value. In other words, the math function will pass that value back to its caller.

The SAS/C Compiler software includes source code in the source directory for the `__matherr` function, so you can change it to do more sophisticated error correction. One typical change is to place a different return value into the exception vector and then return a nonzero action code. This informs the math function that the return value has been changed. You must compile the modified `matherr.c` file and link the resultant object module with your code to incorporate the changes.

Portability UNIX

Returns For the `__matherr` function, a nonzero return indicates that the proposed return value in the exception vector has been changed and that the new value should be used. A return of 0 indicates that the proposed return value is acceptable.

— **__matherr** Math error handler
(continued)

See Also **__except**, **_FPERR**

max Compute the maximum of two values

Synopsis `#include <math.h>`

`v = max(a,b);`

Description This macro computes the maximum of two arithmetic values. It is defined as follows:

```
#define max(a,b) ((a)>(b)?(a):(b))
```

The **max** macro works with any arithmetic type or combination of types.

This function is not available if the `_STRICT_ANSI` flag has been defined.

Portability UNIX

Returns This macro returns the larger of the two parameters.

See Also `min`

mblen Determine the length of a multibyte character

Synopsis `#include <stdlib.h>`

```
length = mblen(s, n);

int length;    /* length or state information */
const char *s; /* pointer to characters or NULL */
size_t n;     /* maximum number of characters to look at */
```

Description This function determines the number of bytes comprising the multibyte character pointed to by the argument *s*, and it is useful for determining how much storage to allocate before calling the `mbstowcs` function.

Portability ANSI

Returns If a `NULL` pointer is passed for the argument *s*, the return value indicates whether the current locale has state-dependent encodings. A nonzero value indicates that it does.

In the Amiga implementation, for all other values for the argument *s*, a 1 is returned since the implementation does not support multibyte characters.

See Also `mbstowcs`, `mbtowc`

mbstowcs Convert a multibyte string to a wide character string

Synopsis `#include <stdlib.h>`

```
length = mbstowcs(pwcs, s, n);
```

```
size_t length; /* length or state information */
wchar_t *pwcs; /* pointer to wide character string */
const char *s; /* pointer to characters or NULL */
size_t n;      /* maximum number of characters to look at */
```

Description This function converts a multibyte string to a wide character string. The **mbstowcs** function for the current locale is passed the input parameters and the result is returned.

Portability ANSI

Returns This function returns the length of the result string.

See Also **mblen**, **mbtowc**

mbtowc Map a multibyte character to a wide character

Synopsis `#include <stdlib.h>`

```
length = mbtowc(pwc, s, n);

int length;      /* length or state information      */
wchar_t *pwc;    /* pointer to wide character      */
const char *s;   /* pointer to characters or NULL  */
size_t n;        /* maximum number of characters to look at */
```

Description This function maps a multibyte character to a wide character. The output buffer must be long enough to hold the result. You can determine the length by calling the `mblen` function.

Portability ANSI

Returns This function returns the length of the multibyte character defined by the locale information. If the argument `s` is `NULL` or if the argument `n` is equal to 0, this function returns 0.

See Also `mblen`, `mbtowc`

memcpy Copy a memory block

Synopsis `#include <string.h>`

```
s = memcpy(to, from, c, n);
```

```
void *s;           /* return pointer */
void *to;          /* destination pointer */
const void *from; /* source pointer */
int c;             /* character value */
unsigned n;        /* number of bytes */
```

Description This function, which was introduced with UNIX System V, copies blocks of memory.

Copying stops once either of these conditions is true:

- ☐ the specified block size has been copied
- ☐ the specified character has been copied.

The **memcpy** function does not handle overlapping memory blocks. If you specify overlapping blocks to this function, the results are unpredictable.

This function neither recognizes nor produces the **NULL** terminator byte usually found at the end of strings.

This function is not available if the **__STRICT_ANSI** flag has been defined.

Portability SAS/C

Returns The **memcpy** function returns a pointer to the character after the argument **c** in the **from** block, or a **NULL** pointer if the **c** argument was not found in the first **n** characters.

See Also **memcpy**, **strcpy**

memchr Find a character in a memory block

Synopsis `#include <string.h>`

```
s = memchr(a,c,n);
```

```
void *s;          /* pointer to character in block */  
const void *a;    /* block pointers */  
int c;            /* character value */  
size_t n;         /* number of bytes */
```

Description This function finds the first occurrence of a character in a block of memory.

This function does not terminate at a **NULL** byte. It always searches **n** bytes.

Portability ANSI

Returns The **memchr** function returns a pointer to the first occurrence of the specified character in the block, or a **NULL** pointer if the character is not found.

`__MemCleanup` Deallocate all allocated memory

Synopsis `#include <stdlib.h>`

`void __stdargs __MemCleanup(void);`

Description The `__MemCleanup` function traverses the linked list of allocated memory to release any memory allocated using SAS/C library functions and not yet returned to the system. No cleanup is performed for memory allocated with Amiga operating system calls.

This function is normally called from the SAS/C startup code as a program is terminating. You can replace the standard `__MemCleanup` function with one of your own.

Portability AmigaDOS

memcmp Compare two memory blocks

Synopsis `#include <string.h>`

```
x = memcmp(a,b,n);

int x;           /* return value */
const void *a,*b; /* block pointers */
size_t n;        /* number of bytes */
```

Description This function compares two blocks of memory, character by character.

The **memcmp** function has a built-in version that is equivalent to the standard library versions. A built-in version generates inline 68000 instructions without needing to make calls to the library. The statement `#include <string.h>` provides a default setting by which any built-in functions are accessed. If you do not want the built-in function, you can use an `#undef memcmp` statement after including the `string.h` file.

This function does not terminate at a **NULL** byte. It always searches *n* bytes.

Portability ANSI

Returns The **memcmp** function returns an integral value as follows:

Return	Meaning
negative	first block sorts below the second
zero	first block equals the second
positive	first block sorts above the second

memcpy Copy a memory block

Synopsis `#include <string.h>`

```
s = memcpy(to,from,n);
```

```
void *s;           /* return pointer */
void *to;          /* destination pointer */
const void *from;  /* source pointer */
size_t n;          /* number of bytes */
```

Description This function, which was introduced with UNIX System V, copies blocks of memory.

The **memcpy** and **movmem** functions are similar, except the former was introduced with UNIX V, while the latter is a traditional SAS/C function. The **memcpy** function does not handle overlapping memory blocks. If you specify overlapping blocks to this function, the results are unpredictable. You may want to use the ANSI function **memmove** instead, since it does handle overlapping blocks.

The **memcpy** function has a built-in version that is equivalent to the standard library versions. A built-in version generates inline 68000 instructions without needing to make calls to the library. The statement `#include <string.h>` provides a default setting by which any built-in functions are accessed. If you do not want the built-in function, you can use an `#undef memcpy` statement after including the `string.h` file.

The **memcpy** function does not place a **NULL** byte at the end of the block, but it always copies `n` bytes.

Portability ANSI

Returns The **memcpy** function returns a pointer to the start of the destination block.

See Also **memccpy**, **memmove**, **movmem**, **strcpy**

memmove Copy bytes in memory

Synopsis `#include <string.h>`

```
p = memmove(dest, from, nbytes);

void *p;                /* same as dest */
void *dest;             /* destination for moved bytes */
const void *from;       /* source of bytes for move */
size_t nbytes;          /* number of bytes to be transferred */
```

Description This function copies the specified number of bytes from one memory location to another. It checks the relative addresses supplied to determine the direction of transfer that will avoid overlap.

Portability ANSI

Returns The **memmove** function returns a pointer to the destination block.

Example

```
/*
 * Make room to insert a word in a character string.
 *
 * This program produces the following output:
 *   This is a test
 *   This is not a test
 */
#include <stdio.h>
#include <string.h>

void main(void)
{
    char string[100];

    strcpy(string, "This is a test");
    printf("%s\n", string);

    /* Shift the words "a test" to make room */
```


memmove Copy bytes in memory
(continued)

```

/* WARNING: Make sure you have plenty of space in */
/* the area you are working with. memmove() and */
/* others do NOT stop at the terminating NULL of */
/* a string, so will blithely write over any */
/* memory you tell them to. This can lead to */
/* different types of problems, from simple */
/* "strange occurrences" to spectacular crashes! */

memmove(string+11,string+7,strlen(string+7)+1);
memcpy(string+7," not ",5);
printf("%s\n",string);
}

```

See Also `memcpy`, `movmem`, `strcpy`

memset Set a memory block to a value

Synopsis `#include <string.h>`

```
s = memset(to,c,n);

void *s;    /* return pointer */
void *to;   /* destination pointer */
int c;      /* character value */
size_t n;   /* number of bytes */
```

Description This function which is compatible with UNIX, sets a block of memory to a value.

The **memset** function has a built-in version that is equivalent to the standard library versions. A built-in version generates inline 68000 instructions without needing to make calls to the library. The statement `#include <string.h>` provides a default setting by which any built-in functions are accessed. If you do not want the built-in function, you can use an `#undef memcmp` statement after including the `string.h` file.

This function neither recognizes nor produces the **NULL** terminator byte usually found at the end of strings.

Portability ANSI

Returns The **memset** function returns a pointer to the destination block.

See Also **setmem**

min Compute the minimum of two values

Synopsis `#include <math.h>`

`v = min(a,b);`

Description This macro computes the minimum of two arithmetic values. It is defined as follows:

```
#define min(a,b) ((a)<=(b)?(a):(b))
```

This macro works with any arithmetic type or combination of types.

This function is not available if the `__STRICT_ANSI` flag has been defined.

Portability UNIX

Returns This macro returns the smallest of the two parameters.

See Also `max`

mkdir Make a new directory

Synopsis `#include <dos.h>`

```
error = mkdir(path);

int error;          /* 0 if successful */
const char *path;   /* points to new directory path string */
```

Description This function makes a new directory in the specified path. For example, if the path is `sys:/abc/def/ghi`, then the new directory is named `ghi` and is in the path `/abc/def` on the volume labeled `sys:.` For AmigaDOS, the path may begin with a drive or volume name and a colon.

Portability UNIX

Returns If the operation is successful, the function returns 0. Otherwise, it returns `-1` and places error information in the external integers `errno` and `_OSERR`.

See Also `errno`, `_OSERR`, `rmdir`

mktime Convert the broken-down time to a `time_t` value

Synopsis `#include <time.h>`

```
t = mktime(ts);
```

```
time_t t;           /* number of seconds since 1/1/70 */
struct tm *ts;      /* broken down time structure */
```

Description This function converts the broken-down time, expressed as local time, to a `time_t` value identical to what the `time` function would return for the specified date and time.

Portability ANSI

Returns The `mktime` function returns the number of seconds since January 1, 1970.

Example

```
/*
 *
 * Get a time value for a very important event
 * Sept 8, 1988 20:16:02
 */
#include <stdio.h>
#include <time.h>

void main(void)
{
    struct tm inputtm;
    time_t    event;

    inputtm.tm_sec = 02;    /* seconds after the minute */
    inputtm.tm_min = 16;    /* minutes after the hour   */
    inputtm.tm_hour = 20;   /* hours since midnight    */
    inputtm.tm_mday = 8;    /* day of the month        */
    inputtm.tm_mon = 9;     /* months since January    */
    inputtm.tm_year = 88;   /* years since 1900        */
    inputtm.tm_isdst = 1;   /* Daylight Savings Time flag */
```

mktime Convert the broken-down time to a `time_t` value
(continued)

```
event = mktime(&inputtm);

printf("%d seconds passed between 1/1/70, 00:00:00"
       " and 9/8/88, 20:16:02\n", event);
}
```

See Also `time`

modf Split a floating-point value

Synopsis `#include <math.h>`

```
x = modf(y,p);

double x; /* fractional part of y */
double y; /* number to be broken up */
double *p; /* integral part of y */
```

Description The **modf** function separates the integral and fractional parts of the argument **y** and returns them as two double-precision floating-point numbers. Both parts have the same sign as the **y** argument. The fractional part is the number that would be obtained by calling the **fmod** functions as follows:

```
x = fmod(y,1.0);
```

Make sure that the second argument of the **modf** function is a pointer to a double-precision floating-point number. Do not use a pointer to an integer.

Portability ANSI

Returns The function return value is the fractional part of the argument **y**, and the integral part is placed in the double-precision floating-point number pointed to by the argument **p**.

Example

```
#include <stdio.h>
#include <math.h>

void modit(double fi)
{
    double ff;

    ff = modf(1.2,&fi);      /* ff contains 0.2 */
    printf("modf(1.2,%lf) = %lf\n", fi, ff);
}

void main(void)
{
    modit(1.0);
}
```

modf Split a floating-point value
(continued)

See Also **fmod**

movmem Move a memory block

Synopsis `#include <string.h>`

```
void movmem(from,to,n);
```

```
const void *from; /* source pointer */
void *to;          /* destination pointer */
unsigned n;        /* number of bytes */
```

Description This function moves blocks of memory. The **movmem** function handles overlapping memory blocks correctly.

This function does not terminate on a **NULL** byte. It always moves exactly **n** bytes.

This function is not available if the **__STRICT_ANSI** flag has been defined.

Portability UNIX

See Also **memmove**

mrnd48 Generate a random long integer (internal seed)

Synopsis `#include <math.h>`

```
z = mrnd48(void);
```

```
long z;          /* random long */
```

Description This function generates random numbers using the linear congruential algorithm and 48-bit arithmetic. The **mrnd48** function uses an internal 48-bit storage area for the seed value.

This function is not available if the `__STRICT_ANSI` flag has been defined.

Portability UNIX

Returns The **mrnd48** function returns signed long integers uniformly distributed over the interval from -2^{31} to $2^{31}-1$.

See Also **jrand48**, **lcong48**, **nrnd48**, **rand**, **srand**

rand48 Generate a random positive long integer (external seed)

Synopsis `#include <math.h>`

```
y = rand48(seed);
```

```
long y; /* random positive long */
unsigned short seed[3]; /* seed value (high bits in seed[0]) */
```

Description This function generates random numbers using the linear congruential algorithm and 48-bit arithmetic. The **rand48** function is provided for cases where several seeds are in use at the same time, so you can specify the seed on each function call.

This function is not available if the **_STRICT_ANSI** flag has been defined.

Portability UNIX

Returns The **rand48** function returns nonnegative long integers uniformly distributed over the interval from 0 to $2^{31}-1$.

See Also **jrand48**, **lcong48**, **lrand48**, **mrnd48**, **rand**, **srand**

offsetof Get the byte offset of a structure member

Synopsis `#include <stddef.h>`

```
size_t offsetof(type, element);
```

Description The `offsetof` macro returns a `size_t` constant specifying the decimal byte offset of a component within a structure. This constant is generated at compile time. Padding for alignment, if any, is included. The operands of the `offsetof` function are a structure type (`type`) and structure member (`element`). The `element` does not include the structure type or the selection operators `.` or `->`.

Portability ANSI

Returns The `offsetof` function returns the byte offset of `element`, within the structure type.

Example This section contains several examples of the use of the `offsetof` macro.

```
#include <stddef.h>

struct AAA { /* Define structure AAA */
    double ddd;
    char ccc;
    int bbb;
};

size_t x;

/* x is the byte offset of component bbb in */
/* struct AAA */
x = offsetof(struct AAA, bbb);
```

offsetof Get the byte offset of a structure member
(continued)

The following example shows a structure, **data**, with an inner structure **base**.

```
#include <stddef.h>

struct data { /* Define struct data */
    int id;
    int *elem;
    char *name;
    struct { /* Define struct type base */
        double proj;
    } base;
};

long ofs;

/* ofs is the byte offset of base.proj */
ofs = offsetof(struct data, base.proj);
```

In the following example, **complex** is defined with a **typedef** statement to be a structure type. The component specification **inner.d[5]** specifies an array element within an inner structure. The variable **y** is set to the offset of the sixth array element in the inner structure (decimal 56).

```
#include <stddef.h>

typedef struct { /* Define struct type complex */
    struct XXX *xptr, *xptr2;
    struct { /* Define struct type inner */
        int count, count2;
        double d[10];
    } inner;
    struct XXX *xptr3;
} complex;

/* y is the byte offset of inner.d[5] */
long y;
y = offsetof(complex, inner.d[5]);
```

offsetof Get the byte offset of a structure member
(continued)

As shown in these examples, the member specification is written as it would be written to access the value of a structure member, except that there is no leading `.` or `->` selection operator.

onbreak Plant a break trap

Synopsis `#include <dos.h>`

```
error = onbreak(func);

int error;          /* 0 if successful */
int (*func)(void); /* pointer to function to be called */
```

Description This function plants a break trap, which is simply a function that gets called whenever the user enters Control-C or Control-D. The standard break keys Control-E and Control-F are ignored by the **onbreak** function.

The **onbreak** function can perform any AmigaDOS operations. If it returns a value of 0, then execution resumes at the interrupted point. Otherwise, the program is aborted immediately.

If the **func** argument is **NULL**, then the current break trap, if any, is removed and the default interrupt handler is restored. With the default handler, Control-C and Control-D cause a requester to appear on the screen with choices to continue or abort.

Detection of Control-C and Control-D is performed during level 1 I/O. Explicit checks for these events can be forced by calls to the function **chkabort**.

Portability AmigaDOS

Returns The **onbreak** function returns 0 if it was successful. The break trap function should return 0 to continue execution and a nonzero value to abort.

Example This program tests the **onbreak** function. After the initial message is printed, you should get the **Break received** message when you press Control-C or Control-D. The second time causes the program to terminate.

```
#include <stdio.h>
#include <dos.h>

int i = 0;
```

onbreak Plant a break trap
(continued)

```
int brk(void)          /* This is the break function */
{
    printf("Break received...\n");
    return(i++);
}

void main(void)        /* This is the main program */
{
    printf("Setting break trap...\n");
    if (onbreak(&brk))
    {
        printf("Can't set break trap\n");
    }
    while(1)
    {
        chkabort();    /* check for CTRL-C          */
    }
}
```

See Also `chkabort`, `signal`

onexit Set an exit trap

Synopsis `#include <stdlib.h>`

```
success = onexit(func);

int success;           /* non-zero for success */
void (*func)(void);    /* trap function pointer */
```

Description This function establishes an exit trap (function) that is called when the program terminates. The trap function is called just before the program returns to the operating system. All buffers are flushed and files are closed before the trap is called. User-allocated memory is not yet freed.

This implementation of the **onexit** function allows only one trap. Each call to the **onexit** function overrides the previous trap. If you call the **onexit** function with a **NULL** pointer, the current trap is removed.

► **Caution** *The exit trap is called after all files have been closed.*
Do not call the **printf** or **cprintf** function from within the exit trap.



Portability ANSI

Returns A zero is returned for success, and a nonzero value is returned if an error is encountered.

Example

```
/* This program tests the onexit function. */

#include <stdlib.h>
#include <stdio.h>

int ex(i)      /* This is the exit trap function */
int i;
{
    WRITE(Output(),"Exit trap hit\n",14);
    return(0);
}

void main(void)      /* This tests the exit trap */
{
    printf("Setting exit trap...\n");
    if(!onexit(ex))
        printf("Can't set trap...\n");
}
```

onexit Set an exit trap
(continued)

```
printf("Exiting with code 2\\n");  
exit(2);  
}
```

See Also **exit**

open Open a level 1 file

Synopsis `#include <fcntl.h>`

```
fh = open(name,mode,prot);

int fh;           /* file handle */
const char *name; /* file name */
int mode;         /* access mode */
int prot;         /* protection mode (O_CREAT only) */
```

Description This function opens a file so that it can be accessed with the level 1 I/O functions. The filename can be any character string that is a valid filename, and it may include a device code and a directory path. The access mode is formed by combining the appropriate symbols using the logical OR operator (|) from the following list of conventional UNIX symbols:

- O_RDONLY** specifies read-only access. No writes are allowed.
- O_WRONLY** specifies write-only access. No reads are allowed.
- O_RDWR** specifies read-write access. Both reads and writes are allowed.
- O_NDELAY** is defined for UNIX compatibility and has no effect under AmigaDOS.
- O_APPEND** is normally used in conjunction with the **O_WRONLY** or **O_RDWR** symbols. It causes the I/O system to seek to the end of the file before the first write operation.
- O_TRUNC** specifies that if the file exists, it is truncated to a length of 0. This flag is normally used with the **O_CREAT**, **O_WRONLY** or **O_RDWR** symbol.
- O_CREAT** specifies that if the file does not already exist, it is created. The protection mode argument is provided for compatibility with existing software, but is ignored under AmigaDOS. To set the protection use the **chmod** function to change the protection bits after the file has been closed.
- O_EXCL** is used only with the **O_CREAT** symbol. If the **O_EXCL** and **O_CREAT** symbols are both present and the file already exists, the **open** function will fail.

The **prot** parameter is only required if the mode is equal to **O_CREAT**. The protection bits are defined in the file `dos/dos.h`.

open Open a level 1 file
(continued)

Portability UNIX

Returns If the operation is successful, the function returns a file handle, which is an integer equal to or greater than 0. Otherwise, it returns `-1` and places error information in the external integers `errno` and `_OSERR`.

See Also `chmod`, `close`, `creat`, `errno`, `_OSERR`

opendir Initiate a directory operation

Synopsis `#include <dir.h>`

```
dfd = opendir(dirname);
```

```
DIR *dfd;           /* return directory file descriptor */
const char *dirname; /* directory name */
```

Description Given a directory name, this routine opens the directory for read access.

Portability UNIX

Returns If successful, this function returns a pointer to a handle that contains the following information:

```
typedef struct _dirdesc {
    long  dd_fd;      /* system directory lock   */
    long  dd_loc;     /* current directory posn */
    long  dd_size;    /* size of dd_buf in bytes */
    char *dd_buf;     /* system structure info  */
} DIR;
```

If it is not successful, this function returns `NULL` pointer.

Example

```
/*
 * An example of opening and searching the contents
 * of a directory for a particular entry.
 */
#include <dir.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int searchdir(char *dname, char *file)
{
    int rc;
    DIR *dfd;      /* directory descriptor */
    struct dirent *dptr; /* dir entry      */

    rc = 0;
    dfd = opendir(dname);

    while ((dptr = readdir(dfd)) != NULL)
```

opendir Initiate a directory operation
(continued)

```

        {
            if (!strcmp(file, dptr->d_name))
            {
                rc = 1; /* Found a match */
                break;
            }
        }
        closedir(dfd);
        return(rc);
    }

void main(int argc, char *argv[])
{
    if (argc < 3)
    {
        printf("Usage: OpenDir <dirname> <filename>\n");
        exit(EXIT_FAILURE);
    }
    if (searchdir(argv[1], argv[2]) == 0)
    {
        printf("File \"%s\" not found in "
               "directory \"%s\"!\n",
               argv[2], argv[1]);
        exit(EXIT_FAILURE);
    }
    printf("Found it!\n");
}

```

See Also `closedir`, `readdir`, `seekdir`, `telldir`

ovlyMgr Overlay manager call point

Synopsis `MOVEQ #ovent,D0`
 `JMP    ovlyMgr`

Description This function is the main entry point to the overlay manager that is called whenever control is to be transferred to an alternate overlay node. The input parameter is an index into the overlay ordinate table that is constructed by the linker. These offsets are automatically assigned by the linker, and all calls to routines across overlay node boundaries are automatically routed through the overlay manager entry point.

Source code for the overlay manager is found in the `ovs.a` file in the source directory.

Calls through the overlay manager destroy registers D0, D1, A0, and A1. Therefore, registerized parameters do not work, and all calls through the overlay manager should use standard stack-based parameter passing (`__stdargs`).

Portability AmigaDOS

perror Print error message

Synopsis `#include <stdio.h>`

```
void perror(s);
```

```
const char *s; /* message prefix */
```

Description This function checks the external integer `errno` and, if it is nonzero, sends an error message to `stderr`. The message consists of the specified prefix, a colon, a space, and the message text from the external array named `__sys_errlist`. This array contains pointers to the various UNIX error messages. The highest error number is given by the contents of the external integer `__sys_nerr`. The SAS/C Compiler software contains the source for these two external items in a file named `syserr` that allows you to modify the messages. The `__sys_nerr` external integer and the `__sys_errlist` external array are declared in the header file `errno.h`.

Portability ANSI

See Also `errno`, `poserr`, `__sys_errlist`, `__sys_nerr`

poserr Print AmigaDOS error message

Synopsis `#include <dos.h>`

```
error = poserr(s);
```

```
int error;          /* contents of _OSERR */
const char *s;      /* message prefix */
```

Description This function checks the external integer `_OSERR` and, if it is nonzero, sends an error message to `stderr`. The message consists of the specified prefix, a colon and space, and the message text from the external array named `__os_errlist`. This array of structures contains pointers to the various AmigaDOS error messages. The highest error number is given by the contents of the external integer `__os_nerr`. The `__os_errlist` external array and the `__os_nerr` external integer are declared in the file `errno.h`. The SAS/C Compiler software contains the source for these two external items in a file named `oserr.c`, which you can modify to customize or expand the messages.

Portability AmigaDOS

Returns The function returns the contents of the external integer `_OSERR` so you can test for an error condition and print a message in one step, as in the example:

```
if(poserr("foo")) goto abort;
```

See Also `_OSERR`, `perror`

printf Formatted print to stdout

Synopsis `#include <stdio.h>`

```
length = printf(fmt,arg1,arg2,...);

int length;          /* number of characters generated */
const char *fmt;     /* format string */
.... argx;           /* arguments */
```

Description This function produces an output stream of ASCII characters and sends the output to `stdout`. `stdout` is usually the user's screen (console).

The `printf` function has a built-in version that is equivalent to the standard library version. The effect of a call to the `printf` function is that the most efficient internal version of the `printf` function is used.

This function works like the `fprintf` function. See the description of the `fprintf` function earlier in this chapter for a complete description.

Portability ANSI

Returns This function returns the number of output characters generated.

Example

```
/*
 * This example prints a message indicating whether
 * the function argument is positive or negative.
 * In the second printf, the width and precision
 * are 15 and 8, respectively.
 */
#include <stdio.h>

void pneg(double value)
{
    char *sign;

    if (value < 0)
    {
        sign = "negative";
    }
    else
    {
        sign = "not negative";
    }

    printf("The number %E is %s.\n",value,sign);
```

printf Formatted print to stdout
(continued)

```
        printf("The number %*.*E is %s.\n", 15, 8, value, sign);
    }

    void main(void)
    {
        pneg(37.8);
        pneg(-18.2);
    }
```

See Also **fprintf**

pow Raise a number to a power

Synopsis `#include <math.h>`

```
r = pow(x,y);
```

```
double r, x, y;
```

Description The **pow** function raises the argument **x** to the **y** power. If **x** is negative and **y** is not an integral value, the `__matherr` function is called with a **DOMAIN** error.

Portability ANSI

See Also `__matherr`, `pow2`

pow2 Raise 2 to a power

Synopsis `#include <math.h>`

```
r = pow2(x);
```

```
double r, x;
```

Description The **pow2** function computes 2^{**x} by calling the **pow** function.

This function is not available if the `__STRICT_ANSI` flag has been defined.

Portability SAS/C

Returns The return value **r** is the value 2^{**x} .

See Also `__matherr`

putc Put a character to a level 2 file

Synopsis `#include <stdio.h>`

```

r = putc(c,fp);

int r;      /* EOF or c */
int c;      /* Character to be output */
FILE *fp;   /* Level 2 file pointer */

```

Description This function puts a single character to the specified level 2 file. The **putc** function is implemented as a macro to maximize execution speed. Therefore, you should not pass expressions that may have side effects to the **putc** function. For example, **putc(c++,fp)** may increment **c** more than once.

Portability ANSI

Returns If successful, this function returns the character to be output; otherwise, it returns **EOF**. For disk files, an **EOF** return usually means that the disk is full. However, this type of return can also occur if the device is write-protected or if a write error occurs. In any case, additional error information can be found in the external integers **errno** and **_OSERR**.

See Also **errno**, **fopen**, **_OSERR**

putchar Put a character to stdout

Synopsis `#include <stdio.h>`

```
r = putchar(c);

int r;      /* EOF or c */
int c;      /* Character to be output */
```

Description This function puts a single character to the level 2 file `stdout`. The `putchar` function is implemented as a macro to maximize execution speed. Therefore, you should not pass expressions that may have side effects to the `putchar` function. For example, `putchar(c++)` may increment `c` more than once.

Portability ANSI

Returns If successful, this function returns the character that was output; otherwise, it returns `EOF`. For disk files, an `EOF` return usually means that the disk is full. However, this type of return can also occur if the device is write-protected or if a write error occurs. In either case, additional error information can be found in the external integers `errno` and `_OSERR`.

See Also `errno`, `fopen`, `_OSERR`

putenv Put a string into the environment

Synopsis `#include <stdlib.h>`

```
error = putenv(env);

int error;          /* 0 if successful */
const char *env;    /* environment string */
```

Description The **putenv** function accepts a string and places it into the current environment. This string has the form:

name=var

If the environment already contains a string beginning with the specific *name* then that string is replaced; otherwise, the new string is added. The text of *var* is written into the file **env:name**.

Environment variables on the Amiga are global so that writing an environment variable from one process sets the variable for all processes.

This function is not available if the **__STRICT_ANSI** flag has been defined.

Portability UNIX

Returns If the **putenv** function is unable to write to the file **env:name**, it returns a nonzero return code.

Example

```
#include <stdio.h>
#include <stdlib.h>

void main(void)
{
    char *e;

    if (putenv("HOCUS=pocus"))          /* Add HOCUS */
    {
        printf("Couldn't add HOCUS\n");
        exit(EXIT_FAILURE);
    }
    printf("Environment variable HOCUS added\n");
```


putenv Put a string into the environment
(continued)

```

e = getenv("HOCUS");           /* Read HOCUS */
if (e == NULL)
{
    printf("Couldn't read HOCUS\n");
    exit(EXIT_FAILURE);
}
printf("Environment variable HOCUS contains: %s\n", e);

printf("Removing environment variable HOCUS\n");
if (putenv("HOCUS="))          /* Remove HOCUS */
{
    printf("Couldn't remove HOCUS\n");
    exit(EXIT_FAILURE);
}
printf("Done.\n");
}

```

See Also `getenv`

putreg Set up 68000-specific registers

Synopsis `#include <dos.h>`

```
void putreg(reg,x);

int reg; /* register number */
long x;  /* value of reg */
```

Description The built-in function **putreg** assigns a value to a register. The valid register values (for the **reg** argument) are defined in the file **dos.h** as follows:

Value	Register Name	Value	Register Name
0	REG_D0	12	REG_A4
1	REG_D1	13	REG_A5
2	REG_D2	14	REG_A6
3	REG_D3	15	REG_A7
4	REG_D4	16	REG_FP0
5	REG_D5	17	REG_FP1
6	REG_D6	18	REG_FP2
7	REG_D7	19	REG_FP3
8	REG_A0	20	REG_FP4
9	REG_A1	21	REG_FP5
10	REG_A2	22	REG_FP6
11	REG_A3	23	REG_FP7

The floating-point registers FP0 through FP7 are available only on Amigas with a math coprocessor. Therefore, you will get an error if you attempt to refer to FP0 through FP7 when the **math=68881** compiler option is active.

putreg Set up 68000-specific registers
(continued)

► **Caution** *Incorrect use of this function can cause serious problems.* ▲

Portability SAS/C

Example `putreg(REG_A4,x); /* set up the current global static base */`

See Also `getreg`

puts Put string to stdout

Synopsis `#include <stdio.h>`

```
error = puts(s);

int error;          /*non-zero if error*/
const char *s;      /*string pointer*/
```

Description The **puts** function copies string **s** to **stdout** (the standard output file). The terminating **NULL** byte is not copied, but a new line character (**\n**) is sent after the string.

Portability ANSI

Returns If an error occurs, the return value is **-1**; otherwise, it is **0**. Additional error information can be found in the external integers **errno** and **_OSERR**.

Example

```
/*
 * This examples writes the following two lines to stdout:
 *
 * This is the first line
 * This is the second line
 */
#include <stdio.h>

void main(void)
{
    puts("This is the first line");
    fputs("This is ",stdout);
    puts("the second line");
}
```

See Also **errno**, **ferror**, **fopen**, **fputc**, **fputs**

qsort Sort a data array

Synopsis `#include <stdlib.h>`

```
void qsort(a,n,size,cmp);

void *a;           /* data array pointer */
size_t n;          /* number of elements in array */
size_t size;       /* element size in bytes */
/* pointer to comparison function */
int (*cmp)(const void *, const void *);
```

Description The **qsort** function sorts the specified array using the ACM 271 algorithm, more popularly known as Quicksort. During its operation, it calls upon the specified comparison routine with pointers to the two array elements being compared. The comparison routine should return an integral result as follows:

Return	Meaning
negative	first element is below the second
positive	first element is above the second
zero	elements are equal

Portability ANSI

raise Generate a signal

Synopsis `#include <signal.h>`

```
ret = raise(sig);

int ret;          /* 0 if successful, nonzero if failed */
int sig;          /* signal to generate                */
```

Description This function simulates the generation of a signal and invokes the proper signal handler for that signal.

Portability ANSI

Returns A nonzero return value indicates failure.

Example

```
/*
 * Cause a floating point overflow
 */
#include <signal.h>

void main(void)
{
    raise(SIGFPE);
}
```

See Also `__matherr`, `onbreak`, `signal`

rand Generate a random number

Synopsis `#include <stdlib.h>`

```
x = rand(void);
```

```
int x;          /* random number */
```

Description The **rand** function returns pseudo-random numbers in the range from 0 to the maximum positive integer value (**RAND_MAX**).

See the **drand48** function and its related functions for more sophisticated random number generation.

Portability ANSI

Returns This function returns an integer value as noted above.

Example

```
/*
 * This example prints 1000 random numbers
 */
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

void main(int argc, char *argv[])
{
    int i, x;

    if (argc > 1)
    {
        stcd_i(argv[1], &x);
        if (x == 0)
        {
            x = 1;
        }
    }
    else
    {
        x=time(NULL);
    }
    srand(x);
    printf("Seed value is %d\n", x);
    printf("Here are 1000 random numbers...\n");
```

rand Generate a random number
(continued)

```
for(i = 0; i < 200; i++)
{
    printf("%5d %5d %5d %5d %5d\n",
        rand(),rand(),rand(),rand(),rand());
}
printf("\n\n");
}
```

See Also **drand48**, **srand**

rawcon Set or unset raw console input

Synopsis `#include <stdio.h>`

```
error = rawcon(flag);

int error;          /* 0 on success */
int flag;           /* non-zero for raw, 0 for non-raw */
```

Description This routine turns on and off the capability of `stdin` that allows you to get single character input without waiting for a new line character.

This routine works with the `getch` and `getchar` functions. Normally, the Amiga console device waits until you press the Return key before it passes the keystrokes you enter to your program. Therefore, the `getchar` function, for example, will not be able to read a single character at a time. Calling `rawcon(1)` forces the console device to pass each character separately. Calling the `rawcon(0)` function restores the console to normal operations.

This function is not available if the `__STRICT_ANSI` flag has been defined.

Portability AmigaDOS

Returns A nonzero return value indicates failure.

Example

```
/*
 *
 * Wait for user to press any key (if possible)
 *
 */
#include <stdio.h>
#include <stdlib.h>

void main(void)
{
    if (!rawcon(1))
    {
        printf("Hit any key to continue\n");
```

rawcon Set or unset raw console input
(continued)

```
        /* make sure output from printf is seen */
        fflush(stdout);
        getchar();
        rawcon(0);
    }

    else
    {
        /* unable to switch the console, wait some other way */
        printf("Sorry, rawcon() didn't work!\n");
        exit(EXIT_FAILURE);
    }
}
```

See Also `getch`, `getchar`

rbrk Release memory

Synopsis `#include <stdlib.h>`

```
error = rbrk(void);
```

```
int error;          /* 0 if successful */
```

Description The **rbrk** function returns all memory in the memory pool, including level 2 I/O buffers, to the operating system.

This function is not available if the `__STRICT_ANSI` flag has been defined.

Portability OLD

Returns This function returns 0 if successful.

See Also `getmem`, `lsbrk`, `malloc`, `sbrk`

read Read from a level 1 file

Synopsis `#include <fcntl.h>`

```
count = read(fh,buffer,length);

int count;           /* actual bytes read or written */
int fh;              /* file handle */
void *buffer;         /* data buffer */
unsigned int length; /* number of bytes to read or write */
```

Description This function reads a level 1 file whose handle was returned by the **creat** or **open** function. Under normal circumstances, the value returned should match the buffer length. If this value is -1 or greater than the requested length, then some type of error occurred, and you should consult the external integers **errno** and **_OSERR**. If the actual length is less than the requested length when reading, this usually means that the file is exhausted. It is a good idea to check the external integers **errno** and **_OSERR** just in case some malfunction caused the short count. Level 1 files are automatically closed by the **exit** and **__exit** functions, which are usually called for you when the program terminates.

Note: When using short integers (**shortintegers** compiler option), the maximum length that can be read is 65,536 (64 megabytes). With normal 32-bit integers, the maximum is 4 gigabytes (**UINT_MAX**).

Portability UNIX

Returns If the operation is successful, the function returns the actual number of bytes transferred. Otherwise, it returns -1 and places error information in the external integers **errno** and **_OSERR**.

See Also **close**, **errno**, **_OSERR**, **open**, **write**

readdir Read a directory element

Synopsis `#include <dir.h>`

```

dptr = readdir(dfd);

struct dirent *dptr; /* pointer to a directory entry */
DIR *dfd;           /* directory file descriptor */

```

Description This function returns the next directory entry for a given directory. The `dfd` argument is the directory file descriptor obtained from a call to the `opendir` function, and the `dptr` argument is the directory entry. A directory entry has the following format:

```

struct dirent {
    u_long d_ino;      /* inode number of entry      */
    u_short d_reclen;  /* length of this record      */
    u_short d_namlen;  /* length of string in d_name */
    off_t d_off;       /* offset of disk directory entry */
    char d_name[MAXNAMLEN+1]; /* maximum length of name */
};

```

Portability UNIX

Returns This function returns a pointer to a `struct dirent`, which is the next directory entry. If the `readdir` function reaches the end of the directory list or if an error occurs, a value of `NULL` is returned. For errors, the error information is placed in the external integers `errno` and `_OSERR`.

See Also `closedir`, `opendir`, `seekdir`, `telldir`

readlocale Read and initialize a new locale

Synopsis `#include <locale.h>`

```
ret = readlocale(locale);
```

```
struct __locale *ret; /* Pointer to the read locale or NULL */
const char *locale;  /* Identifies the type of environment */
```

Description This function reads a new locale and initializes the memory for it. It finds this locale by first examining the **LOCALE** environment variable for the location of a base **LOCALE** directory. It attempts to open a file with the name pointed to by the **locale** argument.

Portability SAS/C

Returns This function returns a value of **NULL** if the locale is missing or invalid.

Example `#include <stdlib.h>`
`#include <locale.h>`

```
void main(void)
{
    struct locale *lcl;
    if (lcl=readlocale(NULL))==NULL)
    {
        printf("Locale environment variable not set.\n");
        exit(1);
    }
    /* locale is set up */
    exit(0);
}
```

See Also `setlocale`

realloc Reallocate memory

Synopsis `#include <stdlib.h>`

```
nb = realloc(b,n);
```

```
void *b;          /* block pointer      */
void *nb;         /* new block pointer */
size_t n;         /* number of bytes   */
```

Description A call to the `realloc` function gets a new block of memory whose size is `n` bytes. Then, it copies the old block `b` to the new block `nb` and releases the old block. If it is less than the old block size, only the first `n` bytes are copied. If `b` is `NULL`, a new block of size `n` is allocated and returned. If `b` is `NULL` and the size is 0, a `NULL` pointer is returned.

Under certain circumstances, you can reallocate a block that has been freed. The `free` function releases a block that was previously obtained with the `calloc`, `malloc`, or `realloc` function. For compatibility with some versions of UNIX, the block is not actually returned to the free space pool until the next time you call the `calloc`, `malloc`, `realloc`, or `free` function. Then, if that next call is to the `realloc` function and the block being reallocated is the one that was just freed, the `realloc` function will proceed correctly. In other words, you can ask the `realloc` function to reallocate a block that was freed as long as you have not called the `calloc`, `malloc`, or `realloc` function in the meantime.

If you are compiling with short integers, the `realloc` function can only allocate 64 megabytes at a time.

This function is called a *level 3 memory allocation* function because it calls on the level 2 function `getmem`. This level is fully compatible with UNIX and with the ANSI standard.

► **Caution** *The level 1 function `rbrk` frees all level 2 and level 3 blocks.* ▲

Portability ANSI

Returns The `realloc` function returns a `NULL` pointer if there is not enough space for the requested block. In this case, the original block `b` is unchanged.

See Also `calloc`, `free`, `getmem`, `malloc`, `rbrk`, `rlsmem`, `sbrk`

remove Remove a file

Synopsis `#include <stdio.h>`

```
error = remove(name);

int error;          /* non-zero if error */
const char *name;   /* file name */
```

Description This function removes the specified file from the system.

The filename argument can include a path, but it cannot include wildcard characters. That is, you can remove only one file at a time.

Portability ANSI

Returns If a nonzero value is returned, some type of error occurred, and additional information can be found in the external integers `errno` and `_OSERR`. The most common errors occur when you try to remove a file that doesn't exist, that is marked as read-only, or is in use.

Example This program removes all files specified in the argument list. It does not allow wildcard characters in the filenames.

```
#include <stdio.h>
#include <stdlib.h>

void main(int argc, char *argv[])
{
    int i;                /* loop counter */
    /* exit code, non-zero if any failures */
    int ret = EXIT_SUCCESS;

    for (i = 1; i < argc; i++)
    {
        if (remove(argv[i]))
        {
            perror("RMV");
            ret = EXIT_FAILURE;
        }
    }
    exit(ret);
}
```


remove Remove a file
(continued)

See Also `errno`, `_OSERR`, `unlink`

rename Rename a file

Synopsis `#include <stdio.h>`

```
error = rename(old,new);

int error;          /* 0 for success, nonzero for error */
const char *old;    /* old file name */
const char *new;    /* new file name */
```

Description This function renames a file, if possible. The old name can include a path, but the new name should not. This function fails if it cannot find the old file or if the new name is the same as an existing file.

Portability ANSI

Returns If the function fails, it returns a nonzero integer and places additional error information into the external integers `errno` and `_OSERR`. If it is successful, it returns a 0.

Example

```
/*
 *
 * This is a version of the rename command
 * that prompts for the old and new names.
 *
 */
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
#include <dos.h>

void main(int argc, char *argv[])
{
    char old[FMSIZE],new[FMSIZE];
    char *pold,*pnew;
```

rename Rename a file
(continued)

```

if (argc < 2)      /* Get old file name */
{
    printf("OLD FILE: ");
    if (fgets(old,sizeof(old),stdin) == NULL)
    {
        exit(EXIT_FAILURE);
    }
    old[strlen(old)-1] = '\0';
    pold = old;
}
else
{
    pold = argv[1];
}

if (argc < 3)
{
    printf("NEW FILE: ");
    if (fgets(new,sizeof(new),stdin) == NULL)
    {
        exit(EXIT_FAILURE);
    }
    new[strlen(new)-1] = '\0';
    pnew = new;
}
else
{
    pnew = argv[2];
}

if (rename(pold,pnew))
{
    perror("RENAME");
    exit(EXIT_FAILURE);
}
}

```

repmem Replicate values through a block

Synopsis `#include <string.h>`

```
void repmem(to,vt,nv,nt);

void *to;           /* destination pointer */
const void *vt;      /* value template */
size_t nv;          /* number of bytes in template */
size_t nt;          /* number of templates in block */
```

Description This function repeats a template of values throughout a block. It is very useful when you need to initialize an array of structures to some nonzero pattern.

This function neither recognizes nor produces the **NULL** terminator byte usually found at the end of strings.

This function is not available if the **__STRICT_ANSI** flag has been defined.

Portability SAS/C

Example

```
/*
 * Initialize an array of structures
 */
#include <stdio.h>
#include <string.h>

struct XMP
{
    int x;
    double d;
};

struct XMP xmp[200];

void initxmp(void)
{
    xmp[0].x=10;
    xmp[0].d=20.0;

    /* All elements are now initialized to 10 and 20.0 */
    repmem(&xmp[1],&xmp[0],sizeof(struct XMP),199);
}
```

repmem Replicate values through a block
(continued)

```
void main(void)
{
    initxmp();

    printf("xmp[0].x = %d, xmp[0].d = %lf\n",
           xmp[0].x, xmp[0].d);
    printf("xmp[199].x = %d, xmp[199].d = %lf\n",
           xmp[199].x, xmp[199].d);
}
```

rewind Reset a level 2 file position to its first byte

Synopsis `#include <stdio.h>`

```
void rewind(fp);
```

```
FILE *fp;    /* file pointer */
```

Description The **rewind** function resets the specified file to its first byte and is equivalent to the following **fseek** call:

```
fseek(fp, 0L, 0);
```

The **rewind** function also clears the error indicator on the file.

The **rewind** function is implemented as a macro that calls the **fseek** function.

Portability ANSI

See Also **errno**, **fopen**, **fseek**, **lseek**, **_OSERR**, **tell**

rewinddir Reset to the start of the directory

Synopsis `#include <dir.h>`

```
void rewinddir(dfd);
```

```
DIR *dfd; /* dir file descriptor */
```

Description This macro changes the position from which the `readdir` function will read the next directory entry, resetting the current position to the start of the directory. The `dfd` argument is a directory file descriptor returned from a successful call to the `opendir` function.

The `rewinddir` function is guaranteed only for the life of the directory file descriptor. If a directory is closed and reopened, the directory entry locations may be invalid.

The `rewinddir` function is implemented as a macro that calls the `seekdir` function as follows:

```
seekdir(dfd, (long)0);
```

This function is not available if the `__STRICT_ANSI` flag has been defined.

Portability UNIX

See Also `closedir`, `opendir`, `readdir`, `seekdir`, `telldir`

rlsmem Release a memory block

Synopsis `#include <stdlib.h>`

```
void rlsmem(p,sbytes);
```

```
void *p;                /* block pointer */
unsigned int sbytes;    /* number of bytes */
```

Description This function releases memory blocks that were previously obtained with the **getmem** or **getml** function. If the number of bytes is less than the value **MELTSIZE** as defined in header file **dos.h**, then it is made equal to **MELTSIZE**.

If you compile with short integers, the **rlsmem** function will free only 64 megabytes or less of memory.

This function is equivalent to the **free** function. The second parameter is ignored. The **rlsmem** function is included for compatibility with previous versions of the compiler.

This function is not available if the **__STRICT_ANSI** flag has been defined.

Portability OLD

See Also **free**, **rlsml**

rlsm1 Release a memory block

Synopsis `#include <stdlib.h>`

```
void rlsm1(p, lbytes);
```

```
void *p;          /* block pointer */
long lbytes;      /* number of bytes */
```

Description This function releases memory blocks that were previously obtained with the `getmem` or `getm1` function.

This function is equivalent to the `free` function. The second parameter is ignored. The `rlsm1` function is included for compatibility with previous versions of the compiler.

This function is not available if the `__STRICT_ANSI` flag has been defined.

Portability OLD

See Also `free`, `rlsmem`

rmdir Remove a directory

Synopsis `#include <dos.h>`

```
error = rmdir(path);

int error;          /* 0 if successful */
const char *path;   /* points to directory path string */
```

Description This function removes an existing directory in the specified path. For example, if the path is `sys:/abc/def/ghi`, then the directory named `ghi` is removed from the path `/abc/def` on the system drive. For AmigaDOS, the path may begin with a drive or volume name and a colon.

Portability UNIX

Returns If the operation is successful, the function returns 0. Otherwise, it returns `-1` and places error information in the external integers `errno` and `_OSERR`.

See Also `errno`, `mkdir`, `_OSERR`

rstmem Reset the memory pool

Synopsis `#include <stdlib.h>`

`void rstmem(void);`

Description It is recommended that you use the **getmem** and **rlsmem** functions or the **malloc** and **free** functions instead of this function. The **rstmem** function is provided for compatibility with previous releases.

This function is not available if the **_STRICT_ANSI** flag has been defined.

Portability OLD

See Also **getmem**, **getml**, **rlsmem**, **rlsml**, **sizmem**

sbrk Allocate memory (unsigned)

Synopsis `#include <stdlib.h>`

```
p = sbrk(sbytes);

void *p;                /* block pointer */
unsigned int sbytes;    /* number of bytes */
```

Description The **sbrk** function is provided for compatibility with previous releases of the compiler. Use the **malloc** function instead.

This function is not available if the `__STRICT_ANSI` flag has been defined.

Portability OLD

Returns If successful, this function returns the address of the block just allocated. If the **sbrk** function fails, it returns value `(char *)(-1)`, which is `-1` cast into a character pointer.

See Also **getmem**, **lsbrk**, **malloc**

scanf Formatted input conversions*Synopsis* `#include <stdio.h>``n = scanf(fmt, arg1, arg2, ...);`

```

int n;                /* number of input items matched, or EOF */
const char *fmt;      /* format string */
---- *argx;           /* pointers to input data areas */

```

Description This function performs formatted input conversions on text obtained from the standard input file.

This function works like the `fscanf` function. See the description of the `fscanf` function earlier in this chapter for a complete description.

Portability ANSI

Returns The function returns the number of assignments that were made. For example, a return value of 3 indicates that conversion results were assigned to the arguments `arg1`, `arg2`, and `arg3`.

See Also `fscanf`, `sscanf`

seed48 Set all 48 bits of an internal seed

Synopsis `#include <math.h>`

```
pseed = seed48(seed);
```

```
unsigned short seed[3]; /* seed value (high bits in seed[0]) */
unsigned short *pseed; /* pointer to internal seed */
```

Description This function generates random numbers using the linear congruential algorithm and 48-bit arithmetic.

The **seed48** function allows you to initialize the internal 48-bit seed to something other than the default. The entire 48 bits are loaded from the specified array.

See the **lcong48** function for additional information.

This function is not available if the `__STRICT_ANSI` flag has been defined.

Portability UNIX

Returns This function returns a pointer to the internal seed array.

See Also **lcong48**, **rand**, **srand**, **srand48**

seekdir Reposition a directory operation

Synopsis `#include <dir.h>`

```
void seekdir(dfd, loc);
DIR *dfd;           /* dir file descriptor */
long loc;           /* entry location */
```

Description This routine changes the position from which the `readdir` function will read the next directory entry. The `dfd` argument is a directory file descriptor returned from a successful call to the `opendir` function. The `loc` argument is a directory entry location as returned from a call to the `telldir` function.

Seek locations are guaranteed only for the life of the directory file descriptor. If a directory is closed and reopened, the directory entry locations may be invalid.

Portability UNIX

See Also `closedir`, `opendir`, `readdir`, `rewinddir`

setbuf Set the buffer mode for a level 2 file

Synopsis `#include <stdio.h>`

```
void setbuf(fp, buff);

FILE *fp;           /* file pointer */
const char *buff;   /* buffer pointer */
```

Description This function sets the buffering mode for a level 2 file. You should call this function after calling the **fopen** function and before calling any other level 2 I/O functions. If you do not call this function in the correct sequence, the file may become corrupted. Do not allocate a buffer on the stack within a function, attach it to a file, and then return from the function. This will corrupt the stack and cause a system failure.

The level 2 I/O system automatically allocates a buffer using the **getmem** function when you perform the first read or write operation. Then, the data being read or written are staged through this buffer to improve I/O efficiency. If you would rather use your own buffer instead of having one allocated for you, call the **setbuf** function with a non-**NULL** buffer pointer. The buffer size must be at least as large as the value given in the external integer `__bufsize`, which defaults to the value of the symbol **BUFSIZ**, defined in the file **stdio.h**.

You can eliminate buffered I/O by calling the **setbuf** function with a **NULL** buffer pointer. When this is done, physical I/O occurs whenever your program performs a level 2 read or write operation, even if only one byte is being transferred. This is very inefficient for disk files but often desirable for terminal or communication ports.

Portability ANSI

See Also **fopen**, **setnbf**, **setvbuf**

setjmp Set long jump parameters

Synopsis `#include <setjmp.h>`

```
ret = setjmp(save);

int ret;          /* return code */
jmp_buf save;     /* address of save area */
```

Description The **setjmp** function saves the current stack mark in a specified save area and returns a code of 0. A subsequent call to the **longjmp** function with the same save area causes control to return to the next statement after the original **setjmp** call. In other words, the statement immediately after the **setjmp** call will be executed twice, once after you call the **setjmp** function and once after you call the **longjmp** function.

Do not use the **longjmp** function after the function calling **setjmp** has returned to its caller. This cannot possibly succeed, because the stack frame for that function no longer exists.

This mechanism is useful for quickly popping up through multiple layers of function calls under exceptional circumstances.

Portability ANSI

Returns A return code of 0 from the **setjmp** function indicates that this is the initial call to save the stack. A nonzero return code indicates that the **longjmp** function has been executed.

Example `#include <stdio.h>`
`#include <setjmp.h>`

```
jmp_buf save;

void foo(void)
{
    longjmp(save, 1);
}

void main(void)
{
    int ret;

    ret=setjmp(save);

    if (ret==0)
```

setjmp Set long jump parameters
(continued)

```
    {  
        /* setjmp has been called, but not longjmp */  
        foo();  
    }  
    else  
    {  
        /* longjmp has been called */  
        printf("all done\n");  
    }  
}
```

See Also `longjmp`

setlocale Set locale information for a program

Synopsis `#include <locale.h>`

```
ret = setlocale(category, locale);

char *ret;          /* Pointer to the selected locale portion */
int category;       /* Names the portion of the locale to be */
                   /* selected */
const char *locale; /* Identifies the type of environment */
```

Description This function selects the appropriate portion of the program's locale as specified by the category and locale arguments. The **category** argument indicates which portion of a program's locale will be affected. You must specify one of the following values:

Value	Portion Affected
LC_COLLATE	the behavior of the strcoll and strxfrm functions
LC_CTYPE	the character-handling and multibyte functions
LC_NUMERIC	the decimal point character for the formatted I/O and string conversion functions, and the nonmonetary formatting information returned by the localeconv function
LC_TIME	the behavior of the strftime function
LC_MONETARY	the monetary formatting information returned by the localeconv function
LC_ALL	the entire program's locale

setlocale Set locale information for a program
(continued)

The **locale** string, which identifies the type of environment to use, may contain one of three special values:

Value	Meaning
C	Use the minimal environment for C translation.
" "	Use the Amiga native environment.
NULL	Use the current default locale without changing it.

If the **locale** argument is not one of these strings, the **setlocale** function searches its internal list of locale environments for a matching one. If it finds one, it uses it. Otherwise, it attempts to open a disk-based locale specification by using the **readlocale** function.

Portability ANSI

Returns If it finds the selected environment, the **setlocale** function returns a pointer to a string associated with the requested category. If it cannot find the environment, it returns a **NULL** pointer, and the program's locale is not changed. This string is considered read-only and is valid until the next call to the **setlocale** function.

See Also **localeconv**, **readlocale**

setmem Set a memory block to a value

Synopsis `#include <string.h>`

```
void setmem(to,n,c);

void *to;      /* destination pointer */
unsigned n;    /* number of bytes */
int c;         /* character value */
```

Description This function sets blocks of memory to a value. It neither recognizes nor produces the **NULL** terminator byte usually found at the end of strings.

This function is similar to the built-in function **memset**, which is an ANSI function.

This function is not available if the **__STRICT_ANSI** flag has been defined.

Portability UNIX

See Also **memset**, **repmem**

setnbf Turn off I/O buffering for level 2 file

Synopsis `#include <stdio.h>`

```
void setnbf(fp);
```

```
FILE *fp; /* file pointer */
```

Description This function eliminates buffered I/O for a level 2 file. You should call this function after calling the **fopen** function and before calling any other level 2 I/O functions. If you do not call this function in the proper sequence, the file may become corrupted.

After you call the **setnbf** function, physical I/O occurs whenever your program performs a level 2 read or write operation, even if only one byte is being transferred. This is very inefficient for disk files but often desirable for terminal or communication ports.

This function is equivalent to:

```
setbuf(fp, NULL);
```

This function is not available if the **__STRICT_ANSI** flag has been defined.

Portability UNIX

See Also **fopen**, **setbuf**, **setvbuf**

setvbuf Set the buffer mode for a level 2 file

Synopsis `#include <stdio.h>`

```
error = setvbuf(fp, buff, type, size);

int error;           /* 0 if successful */
FILE *fp;           /* file pointer */
const char *buff;    /* buffer pointer */
int type;           /* type of buffering */
size_t size;        /* buffer size in bytes */
```

Description This function sets the buffering mode for a level 2 file. You should call this function after calling the `fopen` function and before calling any other level 2 I/O functions. If you do not call this function in the correct sequence, the file may become corrupted. Do not allocate a buffer on the stack within a function, attach it to a file, and then return from this function. This sequence will corrupt the stack and cause a system failure.

The level 2 I/O system automatically allocates a buffer using the `getmem` function when you perform the first read or write operation. Then the data being read or written are staged through this buffer to improve I/O efficiency. If you would rather use your own buffer instead of having one allocated for you, call the `setvbuf` function with a non-NULL buffer pointer. The buffer size must be at least as large as the value given in the external integer `__bufsize`, which defaults to the value of the symbol `BUFSIZ`, defined in the file `stdio.h`.

The `setvbuf` function can set line-buffered mode, attach a buffer of nonstandard size, or turn off buffering. The `type` argument must be one of the following symbols defined in the file `stdio.h`:

Type	Meaning
<code>__IOFBF</code>	fully buffered
<code>__IOLBF</code>	line buffered
<code>__IONBF</code>	nonbuffered

For the `__IOFBF` and `__IOLBF` symbols, the specified buffer is attached to the file unless the `buff` argument is `NULL`, in which case a buffer is automatically allocated on the first read or write. For the `__IONBF` symbol, the `buff` and `size` arguments are ignored.

setvbuf Set the buffer mode for a level 2 file
(continued)

The line-buffered mode is useful for interactive applications. When in this mode, the buffer is flushed whenever a new line is sent, the buffer is full, or input is requested. However, you must use the **fputc** and **fputchar** functions instead of the **putc** and **putchar** macros for line buffering to work correctly. The macros do not check if line-buffered mode is active, so they behave as if the file were fully buffered.

Portability ANSI

Returns The **setvbuf** function returns a nonzero error code if type or size is invalid.

See Also **fopen**, **setbuf**, **setnbf**

signal Establish event traps

Synopsis `#include <signal.h>`

```
oldfun = signal(sig,newfun);

void (*oldfun)(int); /* old trap function */
int sig;             /* signal number */
void (*newfun)(int); /* new trap function */
```

Description This function establishes traps for various events that can occur outside of your program.

The **newfun** argument specifies the action to be taken when the signal occurs, as follows:

SIG_IGN ignore the signal.

SIG_DFL take the system default action, as indicated above for each signal.

If the **newfun** argument is not either of the above, then it must be a valid function pointer. When the signal is detected, the action is reset to either **SIG_DFL** or **SIG_IGN**, depending on the particular signal. Then, the trap function is called with an integer argument specifying which signal was detected (for example, **SIGINT**). The trap function can take whatever action is necessary, including calling the **signal** function again to re-establish itself as the trap function. If the function returns, execution continues at the point in your program where the signal was detected.

The **sig** argument specifies which signal is being trapped, using the following symbols defined in the file **signal.h**:

SIGFPE occurs whenever a floating-point error is detected and the standard version of the **CXFERR** function is installed. If you install your own version, you must duplicate our code (supplied as a file named **CXFERR.C**) to provide this signal.

SIGINT occurs whenever the user operates the Control-C or Control-D key combination. The default action for AmigaDOS is to abort your program. If you specify a function to be called, the signal is reset to **SIG_IGN** when the interrupt occurs. Your function should call the **signal** function again if you want to reinstall the trap.

Portability ANSI

signal Establish event traps
(continued)

Returns If the trap can be established, the **signal** function returns a pointer to the previous handler function. Otherwise, it returns the value **SIG_ERR** and places error information in the external integer **errno**.

sin Sine function

Synopsis `#include <math.h>`

```
r = sin(x);
```

```
double r;    /* result */  
double x;    /* angle */
```

Description The `sin` routine computes the sine of an angle expressed in radians.

Portability ANSI

See Also `cos`, `cosh`, `__matherr`, `sinh`

sinh Hyperbolic sine function

Synopsis `#include <math.h>`

```
r = sinh(x);

double r;    /* result */
double x;    /* angle */
```

Description The `sinh` routine computes the normal hyperbolic sine of an angle. A range error occurs if the hyperbolic sine is too large to be represented by a double-precision floating-point number.

Portability ANSI

See Also `cos`, `cosh`, `__matherr`, `sin`

sizmem Get memory pool size

Synopsis `#include <stdlib.h>`

```
size = sizmem(void);
```

```
long size;
```

Description This function returns the number of unallocated bytes in the current memory pool. This value is the sum of the sizes of all unallocated blocks, so it does not indicate the size of the largest free block.

Also, the value does not indicate the maximum amount of memory that can be allocated. That is, the allocation functions will automatically expand the pool when no block of sufficient size is found in the pool.

This function is not available if the `__STRICT_ANSI` flag has been defined.

Portability SAS/C

See Also `getmem`, `getml`, `rlsmem`, `rlsml`, `rstmem`

sprintf Formatted print to a string

Synopsis `#include <stdio.h>`

```
length = sprintf(s,fmt,arg1,arg2,...);

int length;           /* number of characters generated */
char *s;              /* pointer to a character string */
const char *fmt;      /* format string */
type argx;            /* arguments */
```

Description This function produces an output stream of ASCII characters and sends the output to the storage area whose address is given by the argument `s`. Make sure that this area is large enough to hold the maximum number of characters that might be generated. The `sprintf` function also generates a `NULL` byte to terminate the stored string.

This function works like the `fprintf` function. See the description of the `fprintf` function earlier in this chapter for a complete description.

Portability ANSI

Returns This function returns the number of output characters generated. This number does not include the terminating `NULL` byte.

Example

```
/*
 * This example prints a message indicating whether
 * the function argument is positive or negative.
 * In the second printf, the width and precision
 * are 15 and 8, respectively.
 */
#include <stdio.h>

void pneg(double value)
{
    char *sign, buff[256];

    if (value < 0) sign = "negative";
    else sign = "not negative";

    sprintf(buff,"The number %E is %s.\n",
            value,sign);
    printf(buff);
```

sprintf Formatted print to a string
(continued)

```
        sprintf(buff,"The number %*.*E is %s.\n",
                15,8,value,sign);
        printf(buff);
    }

    void main(void)
    {
        pneg(37.8);
        pneg(-18.2);
    }
```

See Also `fprintf`, `printf`

sqsort Sort an array of short integers

Synopsis `#include <stdlib.h>`

```
void sqsort(sa,n);
```

```
short *sa;      /* pointer to short int array */  
size_t n;       /* number of elements in array */
```

Description The **sqsort** function sorts the specified array of short integers using the ACM 271 algorithm, more popularly known as Quicksort.

This function is not available if the `__STRICT_ANSI` flag has been defined.

Portability SAS/C

See Also **dqsort**, **fqsort**, **lqsort**, **qsort**, **tqsort**

sqrt Square root function

Synopsis `#include <math.h>`

```
r = sqrt(x);
```

```
double r, x;
```

Description The **sqrt** function calculates the square root of a number. The number must be a positive argument. If you supply a negative argument, the **__matherr** function will be called with a **DOMAIN** error.

Portability ANSI

See Also **__matherr**, **pow**, **pow2**

srand Set the seed for the rand function

Synopsis `#include <stdlib.h>`

```
void srand(seed);
```

```
unsigned int seed; /* random number seed */
```

Description The **srand** function resets the random number generator to a new seed value. The initial default seed is 1.

Portability ANSI

```
Example  /*
           * This example prints 1000 random numbers
           */
           #include <stdio.h>
           #include <stdlib.h>

           void main(int argc, char *argv[])
           {
               int i, x;

               if (argc > 1)
               {
                   stcd_i(argv[1], &x);
                   if (x == 0)
                   {
                       x = 1;
                   }
                   printf("Seed value is %d\n", x);
                   srand(x);
               }
               printf("Here are 1000 random numbers...\n");
               for (i = 0; i < 200; i++)
               {
                   printf("%5d %5d %5d %5d %5d\n",
                       rand(), rand(), rand(), rand(), rand());
               }
               printf("\n");
           }
```

srand Set the seed for the rand function
(continued)

See Also `drand48`, `erand48`, `jrand48`, `lrand48`, `nrand48`, `rand`, `srand48`

srand48 Set high 32 bits of an internal seed

Synopsis `#include <math.h>`

```
void srand48(hseed);  
long hseed;    /* high 32 bits of seed value */
```

Description The **srand48** function allows you to initialize the internal 48-bit seed used by the functions **drand48**, **erand48**, **lrand48**, **rand48**, and **jrand48**. The value you specify is copied into the high 32 bits of the seed, and the low 16 bits are set to **0x330E**.

 This function is not available if the **__STRICT_ANSI** flag has been defined.

Portability UNIX

See Also **lcong48**, **rand**, **seed48**, **srand**

sscanf Formatted input conversions*Synopsis* `#include <stdio.h>`

```

n = sscanf(ss,fmt,arg1,arg2,...);

int n;           /* number of input items matched, or EOF */
const char *ss;  /* input string (sscanf only) */
const char *fmt; /* format string */
---- *argx;      /* pointers to input data areas */

```

Description This function performs formatted input conversions on text obtained from a string.

This function works like the `fscanf` function. See the description of the `fscanf` function earlier in this chapter for a complete description.

Portability ANSI

Returns This function returns the number of assignments that were made. For example, a return value of 3 indicates that conversion results were assigned to the arguments `arg1`, `arg2`, and `arg3`.

stacksize Get the current stack size

Synopsis `#include <dos.h>`

```
size = stacksize(void);

size_t size;          /* size of current stack */
```

Description This function calculates the size of the current stack and returns its size in bytes.

Portability SAS/C

Returns This function returns the number of bytes available for the entire stack when the program was started.

Example

```
/*
 * Ensure at least 8K is available for certain
 * operations
 */
#include <stdio.h>
#include <dos.h>

void main(void)
{
    char *amount;

    if (stacksize() > 8000)
    {
        /* Do the operation */
        amount = "More than 8000 bytes";
    }
    else
    {
        /* Tell users it is not safe to do it */
        amount = "8000 bytes or less";
    }
    printf("%s stack available.\n", amount);
}
```

See Also `__stack`

stackused Get the amount of stack in use

Synopsis `#include <dos.h>`

```
size = stackused(void);
```

```
size_t size;           /* amount of current stack in use */
```

Description This function calculates the amount of the current stack currently in use and returns the amount in bytes.

Portability SAS/C

Returns This function returns the number of bytes of stack currently being used.

See Also `__stack`

stat Get information on a file

Synopsis `#include <stat.h>`

```
rc = stat(file, st);

int rc;          /* return code */
const char *file; /* file name */
struct stat *st; /* stat info structure */
```

Description This function obtains information for the given file. If the file is not in the current directory, the file path must be included as part of the filename. Permission to read, write, or execute the file is not required.

This function is provided for compatibility with UNIX. For code that will be used only on the Amiga, use the AmigaDOS function **Examine** instead.

The information is placed into the **stat** structure pointed to by the **st** argument. The structure is defined in the file **stat.h** and contains information as follows:

```
struct stat {
    unsigned short    st_mode;      /* file mode */
    ino_t             st_ino;       /* inode number */
    dev_t             st_dev;       /* file system identifier */
    char              *st_rdev;     /* device identifier (volume name) */
    short             st_nlink;     /* number of links */
    unsigned short    st_uid;       /* file owner user-id */
    unsigned short    st_gid;       /* file group user-id */
    off_t             st_size;      /* file size in bytes */
    time_t            st_atime;     /* time last accessed */
    time_t            st_mtime;     /* time last modified */
    time_t            st_ctime;     /* time last status change */
    short             st_type;      /* Amiga file type */
    char              *st_comment;  /* Amiga file comment */
};
```


stat Get information on a file
(continued)

The following table lists defines that are combined with the logical OR operator to form the `st_mode` field. This list is found in the file `fnctl.h`.

Symbol	Meaning
<code>S_ISCRIPT</code>	The object has its script protection bit set.
<code>S_IPURE</code>	The object is executable.
<code>S_IARCHIVE</code>	The file has its archive bit set.
<code>S_IREAD</code>	The file is readable.
<code>S_IWRITE</code>	The file is writable.
<code>S_IEXECUTE</code>	The file is executable.
<code>S_IDELETE</code>	The file is deletable.

Portability UNIX

Returns If the operation is successful, the function returns 0. Otherwise, it returns `-1` and places error information in the external integers `errno` and `_OSERR`.

See Also `chmod`, `errno`, `fstat`, `_OSERR`

stcarg Get an argument

Synopsis `#include <string.h>`

```
length = stcarg(s,b);

int length;          /* number of bytes in argument */
const char *s;       /* text string pointer */
const char *b;       /* break string pointer */
```

Description This function scans the text string until it finds one of the break characters or the **NULL** terminating byte. While scanning, the **stcarg** function skips over substrings that are enclosed in single or double quotes, and the backslash is recognized as an escape character. Break characters will not be detected if they are quoted or preceded by a backslash.

This function is not available if the **__STRICT_ANSI** flag has been defined.

Portability SAS/C

Returns The function returns a count of the number of characters in the argument **s** up to but not including the break character or **NULL** terminator.

Example

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

void main(void)
{
    char a[256],b[256];
    int x;

    while(1)
    {
        printf("Enter text string: ");
        if (fgets(a,sizeof(a),stdin) == NULL)
        {
            break;
        }
    }
}
```

stcarg Get an argument
(continued)

```

printf("Enter break string: ");
if (fgets(b,sizeof(b),stdin) == NULL)
{
    break;
}
x = stcarg(a,b);
printf("Arg length: %d, Arg text: *s\n",
      x, a);
}
printf("\n");
}

```

See Also stpbrk, strcspn, strpbrk

stccpy Copy one string to another

Synopsis `#include <string.h>`

```
size = stccpy(to,from,n);

int size;           /* number of bytes copied */
char *to;           /* destination pointer */
const char *from;   /* source pointer */
int n;              /* maximum source length */
```

Description This function copies the **NULL**-terminated source string to the destination area. The **stccpy** function writes up to **n** characters to the destination and stops copying at the first occurrence of a **NULL** byte. The **stccpy** function always produces a **NULL**-terminated string.

This function is not available if the **__STRICT_ANSI** flag has been defined.

Portability UNIX

Returns This function returns the actual number of bytes placed in the **to** area, including the **NULL** terminator.

Example

```
/*
 * This example prints: Hello, my name is Flo.
 */
#include <stdio.h>
#include <string.h>

void main(void)
{
    char b[256];

    stccpy(b,"Hello, ",256);
    stccpy(&b[strlen(b)],"my name is ",256-strlen(b));
    stccpy(&b[strlen(b)],"Flo.",256-strlen(b));
    puts(b);
}
```

stcd_i Convert a decimal string to an integer

Synopsis `#include <string.h>`

```
length = stcd_i(in, ivalue);

int length;          /* input length */
const char *in;      /* input string pointer */
int *ivalue;         /* integer value pointer */
```

Description This function scans an input string and converts the leading characters into short integers. The input string must begin with a plus sign (+), minus sign (−), or a decimal digit (0 to 9). The `stcd_i` function stops scanning the input string when it reaches the first invalid character. At that point, the resulting value is stored in the area addressed by the second argument.

This function is not available if the `__STRICT_ANSI` flag has been defined.

Portability SAS/C

Returns This function returns the number of input characters converted. The result is 0 if the first character of the input string is not a valid decimal digit, a plus sign (+), or a minus sign (−). In that case, the conversion result pointed to by the second argument is 0.

Example

```
#include <stdio.h>
#include <string.h>

void main(void)
{
    int x;
    int j;
    char b[80];

    while(1)
    {
        printf("\nEnter a decimal value: ");
        if (fgets(b, sizeof(b), stdin) == NULL)
        {
            break;
        }
    }
}
```

strtol Convert a decimal string to an integer
(continued)

```
        x = strtol(b,&j);  
        printf("strtol: Length %d, Result %d\n",x,j);  
    }  
    printf("\n");  
}
```

stcd_l Convert a decimal string to a long integer

Synopsis `#include <string.h>`

```
length = stcd_l(in,lvalue);

int length;          /* input length */
const char *in;      /* input string pointer */
long *lvalue;        /* long integer value pointer */
```

Description This function scans an input string and converts the leading characters into long integers. The input string must begin with a plus sign (+), minus sign (−), or a decimal digit (0 to 9). The `stcd_l` function stops scanning the input string when it reaches the first invalid character. At that point, the resulting value is stored in the area addressed by the second argument.

This function is not available if the `__STRICT_ANSI` flag has been defined.

Portability SAS/C

Returns This function returns the number of input characters converted. The result is 0 if the first character of the input string is not a valid decimal digit, a plus sign (+), or a minus sign (−). In that case, the conversion result pointed to by the second argument is also 0.

Example

```
#include <stdio.h>
#include <string.h>

void main(void)
{
    int x;
    long j;
    char b[80];

    while(1)
    {
        printf("\nEnter a decimal value: ");
        if (fgets(b,sizeof(b),stdin) == NULL)
        {
            break;
        }
    }
}
```

strtol Convert a decimal string to a long integer
(continued)

```
        x = strtol(b,&j);
        printf("strtol: Length %d, Result %ld\n",x,j);
    }
    printf("\n");
}
```


stcgfe Get the filename extension

Synopsis `#include <string.h>`

```
size = stcgfe(ext,name);

int size;           /* size of result string */
char *ext;          /* extension area pointer */
const char *name;   /* file name pointer */
```

Description This function isolates the extension portion of a filename from the path and node. The *node* is the rightmost portion of the filename that is separated from the rest of the name by a colon or a slash. The *extension* is the final part of the node that begins with a period, and the *path* is the leading part of the name up to the node. The following table contains examples of how you can isolate the parts of a filename using the `stcgfe`, `stcgfn`, and `stcgfp` functions.

Name	Path	Node	Extension
<code>myprog.c</code>	<code>NULL string</code>	<code>myprog.c</code>	<code>c</code>
<code>/abc.dir/def</code>	<code>abc.dir</code>	<code>def</code>	<code>NULL string</code>
<code>/abc.dir/def.ghi</code>	<code>/abc.dir</code>	<code>def.ghi</code>	<code>ghi</code>
<code>df0:yourfile</code>	<code>df0:</code>	<code>yourfile</code>	<code>NULL string</code>
<code>/abc/</code>	<code>/abc</code>	<code>NULL string</code>	<code>NULL string</code>

The maximum number of bytes copied into your array is defined in the file `dos.h` as `FESIZE`. You should provide a buffer at least `FESIZE` bytes long.

This function is not available if the `__STRICT_ANSI` flag has been defined.

Portability SAS/C

Returns The `size` value is the same as would be returned by the `strlen` function. That is, if `size` is 0, then the desired portion of the filename could not be found, and the result area contains a `NULL` string.

stcgfe Get the filename extension
(continued)

Example

```
#include <stdio.h>
#include <string.h>
#include <dos.h>

void main(void)
{
    char file[256],path[FMSIZE],node[FNSIZE],ext[F

    while(gets(file) != NULL)
    {
        stcgfe(ext,file);
        stcgfn(node,file);
        stcgfp(path,file);
        printf("PATH: \"%s\"\\n"
               "NODE: \"%s\"\\n"
               "EXT:  \"%s\"\\n",
               path, node, ext);
    }
}
```

See Also stcgfn, stcgfp, strsfm

stcgfn Get a filename from a path

Synopsis `#include <string.h>`

```
size = stcgfn(node,name);

int size;          /* size of result string */
char *node;        /* node area pointer */
const char *name;  /* file name pointer */
```

Description This function isolates the node portion of a filename from the path and extension. The *node* is the rightmost portion of the filename that is separated from the rest of the name by a colon or a slash. The *extension* is the final part of the node that begins with a period, and the *path* is the leading part of the name up to the node. The following table contains examples of how you can isolate the parts of a filename using the `stcgfe`, `stcgfn`, and `stcgfp` functions.

Name	Path	Node	Extension
<code>myprog.c</code>	<code>NULL string</code>	<code>myprog.c</code>	<code>c</code>
<code>/abc.dir/def</code>	<code>abc.dir</code>	<code>def</code>	<code>NULL string</code>
<code>/abc.dir/def.ghi</code>	<code>/abc.dir</code>	<code>def.ghi</code>	<code>ghi</code>
<code>df0:yourfile</code>	<code>df0:</code>	<code>yourfile</code>	<code>NULL string</code>
<code>/abc/</code>	<code>/abc</code>	<code>NULL string</code>	<code>NULL string</code>

The maximum number of bytes copied into your array is defined in the file `dos.h` as `FESIZE`. You should provide a buffer at least `FESIZE` bytes long.

This function is not available if the `__STRICT_ANSI` flag has been defined.

Portability SAS/C

Returns The `size` value is the same as would be returned by the `strlen` function. That is, if `size` is 0, then the desired portion of the filename could not be found, and the result area contains a `NULL` string.

See Also `stcgfe`, `stcgfp`, `strsfn`

stcgfp Get the file path

Synopsis `#include <string.h>`

```
size = stcgfp(path,name);

int size;           /* size of result string */
char *path;         /* path area pointer */
const char *name;   /* file name pointer */
```

Description This function isolates the path portion of a filename from the node and extension. The *node* is the rightmost portion of the filename that is separated from the rest of the name by a colon, slash, or backslash. The *extension* is the final part of the node that begins with a period, and the *path* is the leading part of the name up to the node. The following table contains examples of how you can isolate the parts of a filename using the **stcgfe**, **stcgfn**, and **stcgfp** functions.

Name	Path	Node	Extension
myprog.c	NULL string	myprog.c	c
/abc.dir/def	abc.dir	def	NULL string
/abc.dir/def.ghi	/abc.dir	def.ghi	ghi
df0:yourfile	df0:	yourfile	NULL string
/abc/	/abc	NULL string	NULL string

The maximum number of bytes copied into your array is defined in the file **dos.h** as **FESIZE**. You should provide a buffer at least **FESIZE** bytes long.

This function is not available if the **__STRICT_ANSI** flag has been defined.

Portability SAS/C

Returns The **size** value is the same as would be returned by the **strlen** function. That is, if **size** is 0, then the desired portion of the filename could not be found, and the result area contains a **NULL** string.

stcgfp Get the file path
(continued)

See Also `stcgfe`, `stcgfn`, `strsfn`

stch_i Convert a hexadecimal string to an integer

Synopsis `#include <string.h>`

```
length = stch_i(in,ivalue);

int length;      /* input length */
const char *in;  /* input string pointer */
int *ivalue;     /* integer value pointer */
```

Description This function scans an unsigned string of hexadecimal digits and converts the leading characters into short integers. The string can contain digits from 0 to 9 and letters from A to F or a to f. The **stch_i** function stops scanning when it reaches the first invalid character. At that point, the resulting value is stored in the area addressed by the second argument.

This function is not available if the `__STRICT_ANSI` flag has been defined.

Portability SAS/C

Returns This function returns the number of input characters converted. The result is 0 if the first character of the input string is not a valid hexadecimal digit. In that case, the conversion result pointed to by the second argument is 0.

Example

```
#include <stdio.h>
#include <string.h>

void main(void)
{
    int x;
    int j;
    char b[80];

    while(1)
    {
        printf("\nEnter a hexadecimal value: ");
        if (fgets(b,sizeof(b),stdin) == NULL)
        {
            break;
        }
    }
}
```

stch_i Convert a hexadecimal string to an integer
(continued)

```
        x = stch_i(b,&j);  
        printf("stch_l: Length %d, Result %x\n",x,j);  
    }  
    printf("\n");  
}
```

stch_l Convert a hexadecimal string to a long integer

Synopsis `#include <string.h>`

```
length = stch_l(in,lvalue);

int length;          /* input length */
const char *in;      /* input string pointer */
long *lvalue;        /* long integer value pointer */
```

Description This function scans an unsigned string of hexadecimal digits and converts the leading characters into long integers. The string can contain digits from 0 to 9 and letters from A to F or a to f. The **stch_l** function stops scanning the input string when it reaches the first invalid character. At that point, the resulting value is stored in the area addressed by the second argument.

This function is not available if the `__STRICT_ANSI` flag has been defined.

Portability SAS/C

Returns This function returns the number of input characters converted. The result is 0 if the first character of the input string is not a valid hexadecimal digit. In that case, the conversion result pointed to by the second argument is 0.

Example

```
#include <stdio.h>
#include <string.h>

void main(void)
{
    int x;
    long j;
    char b[80];

    while (1)
    {
        printf("\nEnter a hexadecimal value: ");
        if (fgets(b,sizeof(b),stdin) == NULL)
        {
            break;
        }
    }
}
```


stch_l Convert a hexadecimal string to a long integer
(continued)

```
        x = stch_l(b,&j);  
        printf("stch_l: Length %d, Result %lx\n",x,j);  
    }  
    printf("\n");  
}
```

stci_d Convert an integer to a decimal string

Synopsis `#include <string.h>`

```
length = stci_d(out, ivalue);

int length;    /* output length */
char *out;     /* output buffer pointer */
int ivalue;    /* integer value */
```

Description This function converts an integer into an ASCII string that is the decimal equivalent of the integer. The output area should be at least 7 bytes long, which is large enough to accommodate the maximum possible string, including the terminating **NULL** byte that this function appends.

The first output character is a minus sign if the input value is negative. No special leading character is generated if the value is positive. Leading zeroes are suppressed, and a single 0 character is generated if the input value is 0.

This function is not available if the **__STRICT_ANSI** flag has been defined.

Portability SAS/C

Returns The return value is the number of characters actually placed into the output area, not including the final **NULL** byte.

Example

```
#include <stdio.h>
#include <string.h>

void main(void)
{
    int i,x;
    char b[7];

    while (1)
    {
        printf("\nEnter an integer: ");
        if (scanf("%d",&i) == EOF)
        {
            break;
        }
    }
}
```

stci_d Convert an integer to a decimal string
(continued)

```
        x = stci_d(b,i);  
        printf("stci_d: Length %d, Result %s\n",x,b);  
    }  
    printf("\n");  
}
```

See Also `stci_h`, `stci_o`, `stcl_d`, `stcl_h`, `stcl_o`, `stcu_d`, `stcul_d`

stci_h Convert an integer to a hexadecimal string

Synopsis `#include <string.h>`

```
length = stci_h(out, ivalue);

int length;    /* output length */
char *out;     /* output buffer pointer */
int ivalue;    /* integer value */
```

Description This function converts an integer into an ASCII string that is the hexadecimal equivalent of the integer. The output area should be at least 5 bytes long, which is large enough to accommodate the maximum possible string, including the terminating **NULL** byte that this function appends.

Leading zeroes are suppressed, and a single 0 character is generated if the input value is 0.

This function is not available if the `__STRICT_ANSI` flag has been defined.

Portability SAS/C

Returns The return value is the number of characters actually placed into the output area, not including the final **NULL** byte.

Example

```
#include <stdio.h>
#include <string.h>

void main(void)
{
    int i,x;
    char b[5];

    while (1)
    {
        printf("\nEnter an integer: ");
        if (scanf("%d",&i) == EOF)
        {
            break;
        }
        x = stci_h(b,i);
        printf("stci_h: Length %d, Result %s\n",x,b);
    }
}
```

stci_h Convert an integer to a hexadecimal string
(continued)

```
        printf("\n");  
    }
```

See Also `stci_d`, `stci_o`, `stcl_d`, `stcl_h`, `stcl_o`, `stcu_d`, `stcul_d`

stci_o Convert an integer to an octal string

Synopsis `#include <string.h>`

```
length = stci_o(out,ivalue);

int length; /* output length */
char *out; /* output buffer pointer */
int ivalue; /* integer value */
```

Description This function converts an integer into an ASCII string that is the octal equivalent of the integer. The output area should be at least 7 bytes long, which is large enough to accommodate the maximum possible string, including the terminating **NULL** byte that this function appends.

Leading zeroes are suppressed, and a single 0 character is generated if the input value is 0.

This function is not available if the `__STRICT_ANSI` flag has been defined.

Portability SAS/C

Returns The return value is the number of characters actually placed into the output area, not including the final **NULL** byte.

Example

```
#include <stdio.h>
#include <string.h>

void main(void)
{
    int i,x;
    char b[7];

    while (1)
    {
        printf("\nEnter an integer: ");
        if (scanf("%d",&i) == EOF)
        {
            break;
        }
        x = stci_o(b,i);
        printf("stci_o: Length %d, Result %s\n",x,b);
    }
    printf("\n");
}
```

stci_o Convert an integer to an octal string
(continued)

See Also `stci_d`, `stci_h`, `stcl_d`, `stcl_h`, `stcl_o`, `stcu_d`, `stcul_d`

stcis Count the number of string characters in the set

Synopsis `#include <string.h>`

```
length = stcis(s,b);

int length;      /* span length in bytes */
const char *s;   /* points to string being scanned */
const char *b;   /* points to character set string */
```

Description This function counts the number of characters at the beginning of the string `s` that are included in the character set specified by the argument `b`. For example, if string `s` is `hello`, and set `b` includes the characters `h`, `e`, and `l`, the `stcis` function returns a 4. The `stcis` function is provided for compatibility with previous versions of the compiler.

This function is not available if the `__STRICT_ANSI` flag has been defined.

Portability SAS/C

Returns This function returns the number of bytes that are in the specified character set. The scan always stops when the `NULL` terminator byte is reached.

Example

```
#include <stdio.h>
#include <string.h>

void main(void)
{
    char s1[256],s2[256];

    while(1)
    {
        printf("\nEnter test string: ");
        if (fgets(s1,sizeof(s1),stdin) == NULL)
        {
            break;
        }
        printf("Enter span string: ");
        if (fgets(s2,sizeof(s2),stdin) == NULL)
        {
            break;
        }
    }
}
```


stc**is** Count the number of string characters in the set
(continued)

```
        printf("stc is: %d\n", stc is(s1,s2));  
    }  
    printf("\n");  
}
```

See Also **stc isn**, **strspn**

stciscn Count the number of string characters not in the set

Synopsis `#include <string.h>`

```
length = stciscn(s,b);

int length;      /* span length in bytes */
const char *s;   /* points to string being scanned */
const char *b;   /* points to character set string */
```

Description This function counts the number of characters at the beginning of the string **s** that are not included in the character set specified by the argument **b**. For example, if string **s** is **hello**, and set **b** includes the characters **h**, **e**, and **l**, the **stciscn** function returns a 0. The **stciscn** function is provided for compatibility with previous versions of the compiler.

This function is not available if the **__STRICT_ANSI** flag has been defined.

Portability SAS/C

Returns This function returns the number of bytes that are not in the specified character set. The scan always stops when the **NULL** terminator byte is reached.

Example

```
#include <stdio.h>
#include <string.h>

void main(void)
{
    char s1[256],s2[256];

    while(1)
    {
        printf("\nEnter test string: ");
        if (fgets(s1,sizeof(s1),stdin) == NULL)
        {
            break;
        }
    }
}
```

stcism Count the number of string characters not in the set
(continued)

```
printf("Enter span string: ");
if (fgets(s2,sizeof(s2),stdin) == NULL)
{
    break;
}
printf("stcism: %d\n",stcism(s1,s2));
}
printf("\n");
}
```

See Also `stcis`, `strcspn`

stcl_d Convert a long integer to a decimal string

Synopsis `#include <string.h>`

```
length = stcl_d(out,lvalue);

int length;      /* output length */
char *out;       /* output buffer pointer */
long lvalue;     /* long integer value */
```

Description This function converts a long integer into an ASCII string that is the decimal equivalent of the integer. The output area should be at least 13 bytes long, which is large enough to accommodate the maximum possible string, including the terminating **NULL** byte that this function appends.

The first output character is a minus sign if the input value is negative. No special leading character is generated if the value is positive. Leading zeroes are suppressed, and a single 0 character is generated if the input value is 0.

This function is not available if the `__STRICT_ANSI` flag has been defined.

Portability SAS/C

Returns The return value is the number of characters actually placed into the output area, not including the final **NULL** byte.

Example `#include <stdio.h>`
`#include <string.h>`

```
void main(void)
{
    int x;
    long l;
    char b[13];

    while (1)
    {
        printf("\nEnter an integer: ");
        if (scanf("%ld",&l) == EOF)
        {
            break;
        }
    }
}
```

stcl_d Convert a long integer to a decimal string
(continued)

```
        x = stcl_d(b,l);
        printf("stcl_d: Length %d, Result %s\n",x,b);
    }
    printf("\n");
}
```

See Also `stci_d`, `stci_h`, `stci_o`, `stcl_h`, `stcl_o`, `stcu_d`, `stcul_d`

stcl_h Convert a long integer to a hexadecimal string

Synopsis `#include <string.h>`

```
length = stcl_h(out,lvalue);

int length;      /* output length */
char *out;       /* output buffer pointer */
long lvalue;     /* long integer value */
```

Description This function converts a long integer into an ASCII string that is the hexadecimal equivalent of the integer. The output area should be at least 9 bytes long, which is large enough to accommodate the maximum possible string, including the terminating **NULL** byte that each function appends.

Leading zeroes are suppressed, and a single 0 character is generated if the input value is 0.

This function is not available if the **__STRICT_ANSI** flag has been defined.

Portability SAS/C

Returns The return value is the number of characters actually placed into the output area, not including the final **NULL** byte.

Example

```
#include <stdio.h>
#include <string.h>

void main(void)
{
    int x;
    long l;
    char b[9];

    while (1)
    {
        printf("\nEnter an integer: ");
        if (scanf("%ld",&l) == EOF)
        {
            break;
        }
        x = stcl_h(b,l);
    }
```

stcl_h Convert a long integer to a hexadecimal string
(continued)

```
        printf("stcl_h: Length %d, Result %s\n",x,b);  
    }  
    printf("\n");  
}
```

See Also stci_d, stci_h, stci_o, stcl_d, stcl_o, stcu_d, stcul_d

stcl_o Convert a long integer to an octal string

Synopsis `#include <string.h>`

```
length = stcl_o(out,lvalue);

int length;      /* output length */
char *out;       /* output buffer pointer */
long lvalue;     /* long integer value */
```

Description This function converts a long integer into an ASCII string that is the octal equivalent of the integer. The length of the output area should be at least 12 bytes long, which is large enough to accommodate the maximum possible string, including the terminating **NULL** byte that this function appends.

Leading zeroes are suppressed, and a single 0 character is generated if the input value is 0.

This function is not available if the **__STRICT_ANSI** flag has been defined.

Portability SAS/C

Returns The return value is the number of characters actually placed into the output area, not including the final **NULL** byte.

Example

```
#include <stdio.h>
#include <string.h>

void main(void)
{
    int x;
    long l;
    char b[13];

    while (1)
    {
        printf("\nEnter an integer: ");
        if (scanf("%ld",&l) == EOF)
        {
            break;
        }
        x = stcl_o(b,l);
```


stcl_o Convert a long integer to an octal string
(continued)

```
        printf("stcl_o: Length %d, Result %s\n",x,b);  
    }  
    printf("\n");  
}
```

See Also stci_d, stci_h, stci_o, stcl_d, stcl_h, stcu_d, stcul_d

stclen Measure the length of a string

Synopsis `#include <string.h>`

```
length = stclen(s);
```

```
size_t length;    /* number of bytes in s (before null) */  
const char *s;
```

Description This function, which is equivalent to the ANSI Standard function **strlen**, returns the number of bytes in the string **s** before the **NULL** terminator byte.

 This function is implemented as a macro and is provided only for compatibility with previous releases.

 This function is not available if the **__STRICT_ANSI** flag has been defined.

Portability **OLD**

Returns The length equals the number of bytes in the string before the **NULL** byte.

See Also **strlen**

stco_i Convert an octal string to an integer

Synopsis `#include <string.h>`

```
length = stco_i(in, ivalue);

int length;          /* input length */
const char *in;      /* input string pointer */
int *ivalue;         /* integer value pointer */
```

Description This function scans an unsigned octal string and converts the leading characters into short integers. The string can consist of octal digits 0 to 7. The `stco_i` function stops scanning the input string when it reaches the first invalid character. At that point, the resulting value is stored in the area addressed by the second argument.

This function is not available if the `__STRICT_ANSI` flag has been defined.

Portability SAS/C

Returns This function returns the number of input characters converted. The result is 0 if the first character of the input string is not a valid octal digit. In that case, the conversion result pointed to by the second argument is 0.

Example

```
#include <stdio.h>
#include <string.h>

void main(void)
{
    int x;
    int j;
    char b[80];

    while(1)
    {
        printf("\nEnter an octal value: ");
        if (fgets(b, sizeof(b), stdin) == NULL)
        {
            break;
        }
        x = stco_i(b, &j);
    }
}
```

stcism Count the number of string characters not in the set

Synopsis `#include <string.h>`

```
length = stcism(s,b);

int length;      /* span length in bytes */
const char *s;   /* points to string being scanned */
const char *b;   /* points to character set string */
```

Description This function counts the number of characters at the beginning of the string **s** that are not included in the character set specified by the argument **b**. For example, if string **s** is **hello**, and set **b** includes the characters **h**, **e**, and **l**, the **stcism** function returns a 0. The **stcism** function is provided for compatibility with previous versions of the compiler.

This function is not available if the **__STRICT_ANSI** flag has been defined.

Portability SAS/C

Returns This function returns the number of bytes that are not in the specified character set. The scan always stops when the **NULL** terminator byte is reached.

Example

```
#include <stdio.h>
#include <string.h>

void main(void)
{
    char s1[256],s2[256];

    while(1)
    {
        printf("\nEnter test string: ");
        if (fgets(s1,sizeof(s1),stdin) == NULL)
        {
            break;
        }
    }
}
```

stco_l Convert an octal string to a long integer

Synopsis `#include <string.h>`

```
length = stco_l(in,lvalue);

int length;          /* input length */
const char *in;      /* input string pointer */
long *lvalue;        /* long integer value pointer */
```

Description This function scans an unsigned octal string and converts the leading characters into long integers. The string can consist of octal digits 0 to 7. The `stco_l` function stops scanning the input string when it reaches the first invalid character. At that point, the resulting value is stored in the area addressed by the second argument.

This function is not available if the `__STRICT_ANSI` flag has been defined.

Portability SAS/C

Returns This function returns the number of input characters converted. The result is 0 if the first character of the input string is not a valid octal digit. In that case, the conversion result pointed to by the second argument is 0.

Example

```
#include <stdio.h>
#include <string.h>

void main(void)
{
    int x;
    long j;
    char b[80];

    while(1)
    {
        printf("\nEnter an octal value: ");
        if (fgets(b,sizeof(b),stdin) == NULL)
        {
            break;
        }
        x = stco_l(b,&j);
    }
```

stco_l Convert an octal string to a long integer
(continued)

```
        printf("stco_l: Length %d, Result %lx\n",x,j);  
    }  
    printf("\n");  
}
```

See Also `stcd_i`, `stcd_l`, `stch_i`, `stch_l`, `stco_i`

stcpm Unanchored pattern match*Synopsis* `#include <string.h>``size = stcpm(string,pattern,match);`

```

int size;           /* size of matching string */
const char *string; /* string to be scanned */
const char *pattern; /* pattern string */
char **match;       /* returns pointer to matching string */

```

Description This function scans a string to find a specified pattern. You can use the following wildcard characters to specify a pattern:

Pattern	Matches
?	any single character
c*	zero or more occurrences of character c
c+	one or more occurrences of character c
\?	a question mark (?)
*	an asterisk (*)
\+	a plus sign (+)

Any other character must match exactly. The following table lists some examples.

Pattern	Matches
abc	only abc
ab*c	ac, abc, or abbc
ab+c	abc, abbc, or abbbc
ab?*c	a string starting with ab and ending in c, for example, abxyzc
ab*c	only ab*c

stcpm Unanchored pattern match
(continued)

The match can occur anywhere in the string.

This function is not available if the `__STRICT_ANSI` flag has been defined.

Portability SAS/C

Returns The function returns the size of the matching string or 0 if there was no match. It also returns a pointer to the beginning of the matching string in the parameter `match`.

Example

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

void main(void)
{
    char s[100],p[100],*r;
    int x;

    while(1)
    {
        printf("\nSearch string => ");
        if (gets(s) == NULL)
        {
            break;
        }
        printf("Pattern          => ");
        if (gets(p) == NULL)
        {
            break;
        }
        x = stcpm(s,p,&r);
        if (x)
        {
            printf("stcpm: size = %d, "
                "match = \"%s.*s\"\n",
                x, x, r);
        }
    }
}
```


stcpm Unanchored pattern match
(continued)

```
        else
        {
            printf("stcpm: no match\n");
            exit(EXIT_FAILURE);
        }
    }
    printf("\n");
}
```

See Also `astcsma`, `stcpma`, `stcsma`

stcpma Anchored pattern match

Synopsis `#include <string.h>`

```
size = stcpma(string, pattern);

int size;           /* size of matching string */
const char *string; /* string to be scanned */
const char *pattern; /* pattern string */
```

Description This function scans a string to find a specified pattern. You can use the following wildcard characters to specify a pattern:

Pattern	Matches
?	any single character
c*	zero or more occurrences of character c
c+	one or more occurrences of character c
\?	a question mark (?)
*	an asterisk (*)
\+	a plus sign (+)

Any other character must match exactly. Some examples are:

Pattern	Matches
abc	only abc
ab*c	ac, abc, or abbc
ab+c	abc, abbc, or abbbc
ab?*c	a string starting with ab and ending in c, for example, abxyzc
ab*c	only ab*c

The match must occur at the beginning of the string.

stcpma Anchored pattern match
(continued)

This function is not available if the `__STRICT_ANSI` flag has been defined.

Portability SAS/C

Returns The function returns the size of the matching string or 0 if there was no match.

Example

```
#include <stdio.h>
#include <string.h>

void main(void)
{
    char s[100],p[100];
    int x;

    while(1)
    {
        printf("\nSearch string => ");
        if (gets(s) == NULL)
        {
            break;
        }
        printf("Pattern          => ");
        if (gets(p) == NULL)
        {
            break;
        }
        x = stcpma(s,p);
        if (x)
        {
            printf("stcpma: size = %d, "
                   "match = \"%.*s\"\n",
                   x, x, s);
        }
        else
        {
            printf("stcpma: no match\n");
        }
    }
}
```

stcpma Anchored pattern match
(continued)

```
        printf("\n");  
    }
```

See Also **astcsma, stcpm, stcsma**

stcsma UNIX string pattern match (anchored)

Synopsis `#include <string.h>`

```
length = stcsma (s,p);
```

```
int length;          /* length of matching string */
const char *s;       /* string being scanned */
const char *p;       /* pattern string */
```

Description The function **stcsma** performs an anchored pattern match of the type used by the UNIX shell. The only meta-characters recognized are the asterisk (*) and the question mark (?). The asterisk matches an arbitrary number of characters, and the question mark matches exactly one character. The pattern must match at the beginning of the supplied string. This function is not available if the `__STRICT_ANSI` flag has been defined.

Portability AmigaDOS

Returns This function returns the length of the matching string or 0 if there was no match.

See Also `astcsma`, `stcpm`, `stcpma`

stcu_d Convert an unsigned integer to a decimal string

Synopsis `#include <string.h>`

```
length = stcu_d(out,uvalue);

int length;           /* output length */
char *out;            /* output buffer pointer */
unsigned uvalue;      /* unsigned value */
```

Description This function converts an unsigned integer into an ASCII string that is the decimal equivalent of the integer. The length of the output area should be at least 6 bytes long, which is large enough to accommodate the maximum possible string, including the terminating **NULL** byte that this function appends.

Leading zeroes are suppressed, and a single 0 character is generated if the input value is 0.

This function is not available if the **__STRICT_ANSI** flag has been defined.

Portability SAS/C

Returns The return value is the number of characters actually placed into the output area, not including the final **NULL** byte.

Example

```
#include <stdio.h>
#include <string.h>

void main(void)
{
    unsigned int i;
    int x;
    char b[13];

    while(1)
    {
        printf("\nEnter an integer: ");
        if (scanf("%u",&i) == EOF)
        {
            break;
        }
        x = stcu_d(b,i);
    }
```

stcu_d Convert an unsigned integer to a decimal string
(continued)

```
        printf("stcu_d: Length %d, Result %s\n",x,b);  
    }  
    printf("\n");  
}
```

See Also `stci_d`, `stci_h`, `stci_o`, `stcl_d`, `stcl_h`, `stcl_o`, `stcul_d`

stcul_d Convert an unsigned long integer to a decimal string

Synopsis `#include <string.h>`

```
length = stcul_d(out,ulvalue);

int length;           /* output length */
char *out;            /* output buffer pointer */
unsigned long ulvalue; /* unsigned long integer value */
```

Description This function converts an unsigned long integer into an ASCII string that is the decimal equivalent of the integer. The output area should be at least 12 bytes long, which is large enough to accommodate the maximum possible string, including the terminating **NULL** byte that each function appends.

Leading zeroes are suppressed, and a single 0 character is generated if the input value is 0.

This function is not available if the `__STRICT_ANSI` flag has been defined.

Portability SAS/C

Returns The return value is the number of characters actually placed into the output area, not including the final **NULL** byte.

Example

```
#include <stdio.h>
#include <string.h>

void main(void)
{
    int x;
    unsigned long i;
    char b[12];

    while(1)
    {
        printf("\nEnter an integer: ");
        if (scanf("%lu",&i) == EOF)
        {
            break;
        }
        x = stcul_d(b,i);
```


stcul_d Convert an unsigned long integer to a decimal string
(continued)

```
        printf("stcul_d: Length %d, Result %s\n",x,b);  
    }  
    printf("\n");  
}
```

See Also stci_d, stci_h, stci_o, stcl_d, stcl_h, stcl_o

stpblk Skip blanks

Synopsis `#include <string.h>`

```
q = stpblk(p);

char *q;          /* updated string pointer */
const char *p;    /* string pointer */
```

Description This function advances the string pointer past blank characters, that is, past all the characters for which the `isspace` function is true.

This function is not available if the `_STRICT_ANSI` flag has been defined.

Portability SAS/C

Returns The function returns a pointer to the next nonblank character. The `NULL` terminator byte is not considered a blank, and so the function will not go past the end of the string.

Example

```
#include <stdio.h>
#include <string.h>

void main(void)
{
    char input[256];

    while(1)
    {
        puts("\nEnter a string with leading blanks...");
        if (gets(input) == NULL)
        {
            break;
        }
        printf("%s\n", stpblk(input));
    }
    printf("\n");
}
```

See Also `stcisc`, `strspn`

stpbrk Find a break character in a string

Synopsis `#include <string.h>`

```
p = stpbrk(s,b);

char *p;           /* points to break character in s */
const char *s;     /* string to be scanned */
const char *b;     /* break characters */
```

Description This function scans string **s** to find the first occurrence of a character from break string **b**.

This function is provided for compatibility with previous releases of the compiler. The ANSI function **strpbrk** is equivalent to the **stpbrk** function.

This function is not available if the **__STRICT_ANSI** flag has been defined.

Portability OLD

Returns If no character from string **b** is found in string **s**, a **NULL** pointer is returned. Otherwise, **p** is a pointer to the first break character.

See Also **strcspn**, **strpbrk**, **strspn**

stpchr Find a character in a string

Synopsis `#include <string.h>`

```
p = stpchr(s,c);

char *p;           /* updated string pointer */
const char *s;     /* input string pointer */
int c;             /* character to be located */
```

Description The **stpchr** function scans the input string to find the first occurrence of the character specified by argument **c**. The **stpchr** function is provided for compatibility with previous releases of the compiler. The ANSI function **strchr** is equivalent to the **stpchr** function.

This function is not available if the **__STRICT_ANSI** flag has been defined.

Portability OLD

Returns The **stpchr** function returns a **NULL** pointer if the input string is empty or if the specified character is not found.

See Also **stpchrn**, **strchr**, **strrchr**

stpchrn Find a character not in a string

Synopsis `#include <string.h>`

```
p = stpchrn(s,c);
```

```
char *p;           /* updated string pointer */
const char *s;     /* input string pointer */
int c;             /* character to be located */
```

Description The **stpchrn** function scans the input string for the first occurrence of some character other than that specified in the **c** argument. This function is provided for compatibility with previous releases of the compiler. The ANSI function **strchr** is equivalent to the **stpchrn** function.

This function is not available if the **__STRICT_ANSI** flag has been defined.

Portability OLD

Returns The **stpchrn** function returns a **NULL** pointer if the input string is empty or consists entirely of character **c**.

See Also **stpchr**, **strchr**, **strchr**

strcpy Copy one string to another

Synopsis `#include <string.h>`

```
np = strcpy(to,from);

char *np;           /* points to end of destination string */
char *to;           /* destination pointer */
const char *from;   /* source pointer */
```

Description This function copies the **NULL**-terminated source string to the destination area. The entire source string is copied, and the resulting destination is always **NULL**-terminated.

This function is not available if the `__STRICT_ANSI` flag has been defined.

Portability SAS/C

Returns The `strcpy` function returns a pointer to the end of the destination string, which is useful when you are building a string from several pieces.

Note: The ANSI `strcpy` function returns the `to` string, but this function returns a pointer to the **NULL** byte after the `to` string.

Example

```
/*
 * This example should print: Hello, my name is Flo.
 */
#include <stdio.h>
#include <string.h>

void main(void)
{
    char b[256],*p;

    p = strcpy(b,"Hello, ");
    p = strcpy(p,"my name is ");
    p = strcpy(p,"Flo.");
    puts(b);
}
```

See Also `stccpy`, `strcpy`, `strncpy`

stptime Convert a date array to a string

Synopsis `#include <string.h>`

```
np = stptime(p,mode,date);

char *np;           /* updated output string pointer */
char *p;            /* output string pointer      */
int mode;           /* conversion mode          */
const char *date;   /* date array, as follows   */
                    /* date[0] => year - 1980   */
                    /* date[1] => month (1 to 12) */
                    /* date[2] => day (1 to 31)  */
```

Description This function converts a 3-byte date array into ASCII or BCD according to the mode argument:

Mode	Format	Type	Length
0	yy mm dd	BCD	3 bytes
1	yy mm dd	ASCII	7 bytes
2	mm / dd / yy	ASCII	9 bytes
3	mm - dd - yy	ASCII	9 bytes
4	MMM d yyyy	ASCII	up to 13 bytes
5	Mm ... m d , yyyy	ASCII	up to 19 bytes
6	dd MMM yy	ASCII	10 bytes
7	dd MMM yyyy	ASCII	12 bytes

In the above formats, **MMM** represents a 3-character month abbreviation in capitals, and **Mm**...**m** represents the full month name (for example, January). The **mm**, **dd**, and **yy** terms are 2-character month, day, and year, respectively, while **d** is the date with the leading zero suppressed. The **yyyy** term is the 4-character year obtained by adding 1980 to the first byte of the date array.

For all modes except 0, a **NULL** byte is appended to the output string.

This function is not available if the **__STRICT_ANSI** flag has been defined.

stptime Convert a date array to a string
(continued)

Portability SAS/C

Returns The function does not make validity checks on the date array. It returns a pointer to the first byte past the generated output. For modes other than 0, this is a pointer to the **NULL** terminator byte.

See Also **getclk, getft, stptime**

stpsym Get the next symbol from a string

Synopsis `#include <string.h>`

```
p = stpsym(s,sym,symlen);

char *p;           /* points to next input character */
const char *s;     /* input string */
char *sym;         /* output string */
int symlen;        /* sizeof(sym) */
```

Description This function breaks out the next symbol from the input string. The first character of the symbol must be alphabetic (upper- or lowercase), and the remaining characters must be alphanumeric. The pointer is not advanced past any initial white space in the input string.

The output string is the `NULL`-terminated symbol and will be an empty string if no symbol is found. If the symbol is longer than `symlen-1`, its excess characters are dropped.

This function is not available if the `__STRICT_ANSI` flag has been defined.

Portability SAS/C

Returns The function returns a pointer to the next character past the symbol.

Example

```
#include <stdio.h>
#include <string.h>

void main(void)
{
    char a[256], b[10];
    char *p;

    while(1)
    {
        printf("\nEnter text string: ");
        if (gets(a) == NULL)
        {
            break;
        }

        p = a;
        while(1)
```

stpsym Get the next symbol from a string
(continued)

```

    {
        p = stpsym(p,b,sizeof(b));
        printf("Symbol:  \n%s\n"
              "Residual: \n%s\n\n",
              b, p);
        if (b[0] == '\0')
        {
            break;
        }
        p++;
    }
    printf("\n");
}

```

See Also stcarg, stpbrk, strcspn, strpbrk

stptime Convert a time array to a string

Synopsis `#include <string.h>`

```
np = stptime(p,mode,time);

char *np;           /* updated output string pointer */
char *p;            /* output string pointer */
int mode;           /* conversion mode */
const char *time;   /* time array, as follows */
                    /* time[0] => hour (0 to 23) */
                    /* time[1] => minute (0 to 59) */
                    /* time[2] => second (0 to 59) */
                    /* time[3] => hundredths (0 to 99) */
```

Description This function converts a 4-byte time array into ASCII or BCD according to the mode argument:

Mode	Format	Type	Length
0	hhmmssdd	BCD	4 bytes
1	hhmmss	ASCII	7 bytes
2	hh:mm:ss	ASCII	9 bytes
3	hhmmssdd	ASCII	9 bytes
4	hh:mm:ss.dd	ASCII	12 bytes
5	hh:mm	ASCII	6 bytes
6	hr:mm:ss HH	ASCII	12 bytes
7	hr:mm HH	ASCII	9 bytes

The **hh**, **mm**, **ss**, and **dd** terms are the 2-digit (BCD or ASCII) equivalents of the binary values in the time array. The **hr** term is the 2-digit hour using the 12-hour form, and the **HH** term is either AM or PM.

A **NULL** terminator byte is appended to the ASCII output strings.

This function is not available if the **_STRICT_ANSI** flag has been defined.

stptime Convert a time array to a string
(continued)

Portability SAS/C

Returns The function does not make validity checks on the time array. It returns a pointer to the first byte past the generated output. For modes other than 0, this is a pointer to the **NULL** terminator byte.

See Also **getclk**, **getft**, **stpdate**

stptok Get the next token from a string

Synopsis `#include <string.h>`

```
p = stptok(s,tok,toklen,brk);

char *p;           /* points to next character after token */
const char *s;     /* points to input string */
char *tok;         /* points to output buffer */
int toklen;        /* size of buffer pointed to by tok */
const char *brk;   /* break string */
```

Description This function breaks out the next token from the input string and moves it to the token buffer with a `NULL` terminator. A token consists of all characters in the input string `s` up to but not including the first character that is in the break string. In other words, the `brk` argument specifies the characters that cannot be included in a token. If the input string begins with a break character, then the token buffer will contain a `NULL` string, and the return pointer `p` will be the same as the argument `s`. If no break character is found after `toklen-1` input characters have been moved to the token buffer, or if the input string terminator (a `NULL` byte) is reached, then the scan stops as if a break character were reached.

This function is not available if the `__STRICT_ANSI` flag has been defined.

Portability SAS/C

Returns The function returns a pointer to the next character in the input string.

Example

```
/*
 * This example breaks out words that are
 * separated by blanks or commas.
 * The token buffer takes on the following
 * values as the program loops:
 *
 *   LOOP   TOKEN
 *   ----   -
 *   1      first
 *   2      second
 *   3      third
 *   4      fourth
 */
```

stptok Get the next token from a string
(continued)

```
#include <stdio.h>
#include <string.h>

char test[] = "first, second third, fourth";

void main(void)
{
    char *p = test;
    char token[50];

    while(1)
    {
        p = stptok(p,token,sizeof(token)," ,");
        printf("%s\n",token);
        if (*p == '\0')
        {
            break;
        }
        p = stpblk(++p);
    }
}
```

See Also stpblk, strtok

strbpl Build a string pointer list*Synopsis* `#include <string.h>`

```

n = strbpl(s,max,t);

int n;                /* number of pointers */
char *s[];            /* pointer to string pointer list */
int max;              /* maximum number of pointers */
const char *t;        /* text pointer */

```

Description This function constructs a list of pointers to the strings contained within the specified text array. Each string must be **NULL**-terminated, and the text array must be terminated by a **NULL** string. In other words, array **t** must end with two **NULL** bytes, one to terminate the final string and another to terminate the array. The string pointer list **s** is terminated by a **NULL** pointer.

This function is not available if the `__STRICT_ANSI` flag has been defined.

Portability SAS/C

Returns The return value indicates how many string pointers were placed into array **s**. If the number of strings plus the final **NULL** pointer is greater than the argument **max**, a value of `-1` is returned.

Example

```

#include <stdio.h>
#include <string.h>

char text[] = {"string 1","string 2",'\\0'};

void main(void)
{
    char *list[5];
    int count, i;

```

strbpl Build a string pointer list
(continued)

```

/*
 * The following call has the following effect:
 *
 *      Return value (count) is 2.
 *      list[0] => "string 1"
 *      list[1] => "string 2"
 *      list[2] => NULL
 *
 */
count = strbpl(list,5,text);
printf("%d strings were found:\n", count);
for (i=0; i<count; i++)
{
    printf("%s\n", list[i]);
}
}

```

See Also `getfnl`, `strsrt`

strcat Concatenate strings

Synopsis `#include <string.h>`

```
p = strcat(to,from);

char *p;           /* same as destination string pointer */
char *to;          /* destination string pointer */
const char *from;  /* source string pointer */
```

Description This function concatenates the source string to the end of the destination string, overwriting the existing **NULL** byte at the end of the destination string. The **strcat** function places a **NULL** byte at the new end of the destination string.

Portability ANSI

Returns This function returns a pointer that is the same as the first argument.

Example

```
#include <stdio.h>
#include <string.h>

char first[100];
char *second=" a test";

void main(void)
{
    strcpy(first, "This is");
    strcat(first, second);
    printf("%s\n", first);
    /* output is "This is a test" */
}
```

See Also `strcpy`, `strncat`

strchr Find a character in a string

Synopsis `#include <string.h>`

```
p = strchr(s,c);

char *p;          /* updated string pointer */
const char *s;    /* input string pointer */
int c;            /* character to be located */
```

Description The `strchr` function scans the input string to find the first occurrence of the character specified by the argument `c`.

Portability ANSI

Returns A `NULL` pointer is returned if the input string is empty or if the specified character is not found. Otherwise, this function returns a pointer to the first matching character in the argument `s`.

Example

```
#include <stdio.h>
#include <string.h>

void main(void)
{
    char *p;

    p=strchr("This is a test",'t');

    /* p now points to "test" */
    printf("%s\n",p);

    p=strchr("This is a test",'s');

    /* p now points to "s is a test" */
    printf("%s\n",p);
}
```

See Also `stpchrn`, `strchr`, `strrchr`

strcmp Compare strings, case sensitive

Synopsis `#include <string.h>`

```
x = strcmp(a,b);
```

```
int x;                /* comparison result */
const char *a,*b;     /* strings being compared */
```

Description This function compares two **NULL**-terminated strings. The ASCII collating sequence is used in all cases.

The relative collating sequence of the strings is indicated by the sign of the return value, as follows:

Sign	Meaning
negative	first string is below the second
zero	strings are equal
positive	first string is above the second

If the strings have different lengths, the shorter one is treated as if it were extended with zeroes. The **strcmp** function has a built-in version that is equivalent to the standard library version. The statement `#include <string.h>` provides a default setting by which built-in functions are accessed. If you do not want the built-in function, you can enter an `#undef strcmp` after including the **string.h** file.

Portability ANSI

Returns The sign of the return value indicates the relative collating sequence of the strings, as indicated above.

Example

```
#include <stdio.h>
#include <string.h>

void result(char *name, int r)
```

strcmp Compare strings, case sensitive
(continued)

```

    {
        char *p;

        if (r == 0)
        {
            p = "is equal to";
        }
        if (r < 0)
        {
            p = "is less than";
        }
        if (r > 0)
        {
            p = "is greater than";
        }
        printf("%s string A %s string B\n",name,p);
    }

void main(void)
{
    char a[256],b[256];

    while(1)
    {
        printf("Enter string A: ");
        if (fgets(a,sizeof(a),stdin) == NULL)
        {
            break;
        }
        printf("Enter string B: ");
        if (fgets(b,sizeof(b),stdin) == NULL)
        {
            break;
        }
        result("strcmp: ",strcmp(a,b));
    }
    printf("\n");
}

```

strcmp Compare strings, case sensitive
(continued)

See Also `strcmpi`, `stricmp`, `strncmp`, `strnicmp`

strcmpi Compare strings, case insensitive

Synopsis `#include <string.h>`

```
x = strcmpi(a,b);

int x;                /* comparison result */
const char *a,*b;     /* strings being compared */
```

Description This function compares two NULL-terminated strings using the ASCII collating sequence. The **strcmpi** function does not distinguish between uppercase and lowercase. This function is a hold-over from various Microsoft compilers. Use the **stricmp** function in new code.

The relative collating sequence of the strings is indicated by the sign of the return value, as follows:

Return	Meaning
negative	first string is below the second
zero	strings are equal
positive	first string is above the second

If the strings have different lengths, the shorter one is treated as if it were extended with zeroes.

This function is not available if the **__STRICT_ANSI** flag has been defined.

Portability OLD

Returns The sign of the return value indicates the relative collating sequence of the strings, as indicated above.

See Also **strcmp**, **stricmp**, **strncmp**, **strnicmp**

strcoll Compare strings based on locale

Synopsis `#include <string.h>`

```
res = strcoll(s1, s2);

int res;           /* result of comparison */
const char *s1,*s2; /* strings to compare   */
```

Description This function compares two strings based on the current locale. Because of the limitations of the ANSI specifications, this function cannot correctly handle the rules of the German collating sequence completely.

Portability ANSI

Returns This function returns a 0 if both strings are equal. If string `s1` is logically less than string `s2`, the return value is less than 0. If string `s1` is logically greater than string `s2`, the return value is greater than 0.

Example

```
if (!strcoll("string1", "string2"))
    printf("This is funny, they shouldn't match\n");
```

See Also `setlocale`

strcpy Copy one string to another

Synopsis `#include <string.h>`

```
p = strcpy(to,from);

char *p;           /* same as destination pointer */
char *to;          /* destination pointer */
const char *from;  /* source pointer */
```

Description This function copies the entire **NULL**-terminated source string to the destination area. The resulting destination is always **NULL**-terminated.

The **strcpy** function has a built-in version that is equivalent to the standard library version. The statement `#include <string.h>` provides a default setting by which built-in functions are accessed. If you do not want the built-in function, you can use an `#undef strcpy` statement after including the file **string.h**.

Portability ANSI

Returns This function returns a pointer that is the same as the destination pointer.

See Also **stccpy**, **stpcpy**, **strncpy**

strcspn Count the number of string characters not in the set

Synopsis `#include <string.h>`

```
length = strcspn(s,b);

size_t length;      /* span length in bytes */
const char *s;      /* points to string being scanned */
const char *b;      /* points to character set string */
```

Description This function measures the number of characters at the beginning of input string *s* that are not in the character set specified by the argument *b*.

Portability ANSI

Returns This function returns the number of bytes that are not in the specified character set. The scan always stops when the **NULL** terminator byte is reached.

Example

```
#include <stdio.h>
#include <string.h>

char *test = "This is a test";

void checkspan(char *string, char *set)
{
    printf("String:  %s\nScan Set: %s\n"
           "strcspn:  %d\n", string, set,
           strcspn(string, set));
}

void main(void)
{
    checkspan(test,"xyz");
    checkspan(test,"s");
    checkspan(test,"TXI");
}
```

See Also `stcis`, `stciscn`, `strspn`

strdup Duplicate a string

Synopsis `#include <string.h>`

```
p = strdup(s);
```

```
char *p;           /* points to duplicate string */  
const char *s;     /* points to string being duplicated */
```

Description This function creates a duplicate of the specified string by using the `malloc` and `strcpy` functions to allocate space and copy the string to it.

This function is not available if the `__STRICT_ANSI` flag has been defined.

Portability XENIX

Returns A `NULL` pointer is returned if the `malloc` functions fails. Otherwise, the function returns a pointer to the duplicate string.

See Also `malloc`, `strcpy`

strerror Print the text for a given error number

Synopsis `#include <string.h>`

```
p = strerror(error);

char *p;           /* Pointer to text string */
int error;         /* Error number          */
```

Description This function takes a specified error number and returns a pointer to a text string that describes the error.

Portability ANSI

Returns This function returns a pointer to the text of the corresponding error message. If it could not find the error, it returns a `NULL` pointer. The string is valid until the next call to the `strerror` function and must not be modified by the caller.

Example

```
/*
 * Print out an error from unlinking a file
 */
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

extern long errno;

void main(void)
{
    unlink("xyzzzy:lambda");
    if (errno)
    {
        printf("Error removing file: %s\n",
               strerror(errno));
        exit(EXIT_FAILURE);
    }
}
```

See Also `errno`

strftime Format a time string

Synopsis `#include <time.h>`

```
ret = strftime(s, maxsize, format, timeptr);

size_t ret;                /* number of characters in      */
                           /* formatted string          */
char *s;                   /* string to place characters in */
size_t maxsize;            /* maximum string size        */
const char *format;        /* format instructions for string */
const struct tm *timeptr;  /* broken-down time information */
```

Description This function is similar to the `sprintf` function but has its own formatting instructions for printing out time information.

This function places characters into the array pointed to by the argument `s` in the format specified by the string pointed to by the argument `format`. The `format` argument consists of zero or more conversion specifiers and ordinary characters. All ordinary characters, including the terminating `NULL` character, are copied unchanged into the array, but the conversion specifiers are replaced by the appropriate characters. A conversion specifier consists of a percent (%) character followed by a character. The following list describes the characters with which each conversion specifier is replaced.

Conversion Specifier	Replaced with . . .
%a	the locale's abbreviated weekday name
%A	the locale's full weekday name
%b	the locale's abbreviated month name
%B	the locale's full month name
%c	the locale's appropriate date and time representation
%d	the day of the month as a decimal number (01-31)
%H	the hour (24-hour clock) as a decimal number (00-23)
%I	the hour (12-hour clock) as a decimal number (00-12)

(continued)

strftime Format a time string
(continued)

Conversion Specifier	Replaced with . . .
%j	the day of the year as a decimal number (001-366)
%m	the month as a decimal number (01-12)
%M	the minute as a decimal number (00-59)
%p	the locale's equivalent of the AM and PM designations associated with a 12-hour clock
%S	the second as a decimal number (00-61)
%U	the week number of the year (the first Sunday as the first day of week 1) as a decimal number (00-53)
%w	the weekday as a decimal number (0-6), where Sunday is 0
%W	the week number of the year (the first Monday as the first day of week 1) as a decimal number (00-53)
%x	the locale's appropriate date representation
%X	the locale's appropriate time representation
%Y	the year without the century as a decimal number (00-99)
%Y	the year with the century as a decimal number
%Z	the time zone name or abbreviation; no characters indicates the time zone is not determinable
%%	a percent sign.

No more than **maxsize** characters are placed into the array. The appropriate characters are determined by the **LC_TIME** category of the current locale and by the values contained in the structure pointed to by the argument **timeptr**. If copying takes place between objects that overlap or if the conversion specifier is not one of those listed above, the behavior is undefined.

Portability ANSI

strftime Format a time string
(continued)

Returns This function returns the number of characters placed into the string pointed to by the argument **s**, not including the terminating **NULL** character. Otherwise, the **strftime** function returns a 0, and the contents of the **s** argument are indeterminate.

See Also **asctime, gmtime, localtime, setlocale**

stricmp Compare strings, case insensitive

Synopsis `#include <string.h>`

```
x = stricmp(a,b);
```

```
int x;                /* comparison result */
const char *a,*b;     /* strings being compared */
```

Description This function compares two **NULL**-terminated strings using the ASCII collating sequence, but does not distinguish between uppercase and lowercase.

The relative collating sequence of the strings is indicated by the sign of the return value, as follows:

Return	Meaning
negative	first string is below the second
zero	strings are equal
positive	first string is above the second

This function is not available if the `__STRICT_ANSI` flag has been defined.

Portability SAS/C

Returns The sign of the return value indicates the relative collating sequence of the strings, as indicated above.

Example

```
#include <stdio.h>
#include <string.h>

void result(char *name, int r)
{
    char *p;

    if (r == 0)
    {
        p = "is equal to";
    }
}
```

stricmp Compare strings, case insensitive
(continued)

```

        if (r < 0)
        {
            p = "is less than";
        }
        if (r > 0)
        {
            p = "is greater than";
        }
        printf("%s string A %s string B\n",name,p);
    }

void main(void)
{
    char a[256], b[256];

    while(1)
    {
        printf("Enter string A: ");
        if (fgets(a,sizeof(a),stdin) == NULL)
        {
            break;
        }
        printf("Enter string B: ");
        if (fgets(b,sizeof(b),stdin) == NULL)
        {
            break;
        }
        result("stricmp: ",stricmp(a,b));
    }
    printf("\n");
}

```

See Also **strcmp, strncmp, strnicmp**

strins Insert a string

Synopsis `#include <string.h>`

```
void strins(to,from);
```

```
char *to;           /* destination string */
const char *from;   /* source string      */
```

Description This function inserts the source string in front of the destination. Both strings must be **NULL**-terminated, and the destination is shifted to the right (upward in memory) to accommodate the source string. The final result is a single **NULL**-terminated string.

Make sure that the destination area is large enough to hold both strings.

This function is not available if the `__STRICT_ANSI` flag has been defined.

Portability SAS/C

Example

```
#include <stdio.h>
#include <string.h>

void main(void)
{
    char *here = "Here ";
    char now[30];

    strcpy(now, "and now");
    printf("%s, %s\n", here, now);
    strins(now, here);          /* now => "Here and now" */
    printf("%s\n",now);
}
```

See Also `strcat`

strlen Measure the length of a string

Synopsis `#include <string.h>`

```
length = strlen(s);
```

```
size_t length; /* number of bytes in s (before null) */
const char *s;
```

Description This function returns the number of bytes in string **s** before the **NULL** terminator byte.

The **strlen** function has a built-in version that is equivalent to the standard library version. The statement `#include <string.h>` provides a default setting by which built-in functions are accessed. If you do not want the built-in function, you can enter an `#undef strlen` statement after including the file **string.h**.

Portability ANSI

Returns This function returns the number of bytes in the string before the **NULL** byte.

Example

```
x = strlen("abc"); /* x is 3 */
x = strlen("");   /* x is 0 */
```

See Also **stclen**

strlwr Convert a string to lowercase

Synopsis `#include <string.h>`

```
p = strlwr(s);

char *p;    /* return pointer (same as s) */
char *s;    /* string pointer */
```

Description This function converts all alphabetic characters in the specified **NULL**-terminated string to lowercase, according to the 7-bit ASCII collating sequence.

This function is not available if the **__STRICT_ANSI** flag has been defined.

Portability XENIX

Returns This function returns the original string pointer.

See Also **stricmp, strupr, tolower, toupper**

strmfe Make a filename with an extension

Synopsis `#include <string.h>`

```
void strmfe(newname,oldname,ext);

char *newname;          /* new file name */
const char *oldname;    /* old file name */
const char *ext;        /* extension */
```

Description This function copies the old filename to the new name, deleting any extension. Then it appends the specified extension to the new filename, with an intervening period. For example:

Old name	Extension	New name
<code>df1:myprog.c</code>	<code>cc</code>	<code>df1:myprog.cc</code>
<code>abc</code>	<code>o</code>	<code>abc.o</code>

The **newname** area must be large enough to accept the filename string and the separator. A safe size is **FMSIZE** which is defined in the **dos.h** header file.

This function is not available if the **__STRICT_ANSI** flag has been defined.

Portability SAS/C

See Also `strmfnc`, `strmfp`

strmf Make a filename from components

Synopsis `#include <string.h>`

```
void strmf(file,drive,path,node,ext)
```

```
char *file;           /* file name pointer */
const char *drive;    /* drive code pointer */
const char *path;     /* directory path pointer */
const char *node;     /* node pointer */
const char *ext;      /* extension pointer */
```

Description This function makes a filename from four possible components. The name is constructed as follows:

```
drive:path/node.ext
```

If the `drive` pointer is not `NULL`, the `drive` pointer is moved to the area pointed to by the `file` argument. Then, a colon is inserted unless one is already there. Next, if the `path` pointer is not `NULL`, it is appended to `file`, and the directory separator specified by `__SLASH` is added if necessary. The `node` string is appended next, unless it is `NULL`. Finally, if the `ext` pointer is not `NULL`, a period is appended to `file`, followed by the `ext` string.

Make sure that the `file` pointer refers to an area which is large enough to hold the result.

This function is not available if the `__STRICT_ANSI` flag has been defined.

Portability SAS/C

Example

```
#include <stdio.h>
#include <string.h>
#include <dos.h>

char buffer[FMSIZE];

void main(void)
{
    /* The next statements both place "abc/def/ghi" */
    /* into the buffer. */

    printf("", 'abc/def', 'ghi', '\n');
```

strmf Make a filename from components
(continued)

```

strmf(buffer, NULL, "abc/def", "ghi", NULL);
printf("result = %s\n\n", buffer);

printf('', 'abc/def/', 'ghi', '\n');
strmf(buffer, NULL, "abc/def/", "ghi", NULL);
printf("result = %s\n\n", buffer);

/* The next statements both generate "df0:myfile.str" */

printf('df0', '', 'myfile', 'str\n');
strmf(buffer, "df0", NULL, "myfile", "str");
printf("result = %s\n\n", buffer);

printf('df0:', '', 'myfile', 'str\n');
strmf(buffer, "df0:", NULL, "myfile", "str");
printf("result = %s\n\n", buffer);
}

```

See Also strmf, strmf

strmfp Make a filename from the path or node

Synopsis `#include <string.h>`

```
void strmfp(name,path,node);
```

```
char *name;           /* file name */
const char *path;     /* directory path */
const char *node;     /* node */
```

Description This function copies the **path** string (if it is not **NULL**) to the file **name** area and appends the **__SLASH** separator if the **path** string does not end with a slash or colon. Then, the **node** string is appended to the file **name**. **__SLASH** is an external character variable that defaults to a slash (/).

The **name** area must be large enough to accept the filename string.

This function is not available if the **__STRICT_ANSI** flag has been defined.

Portability SAS/C

See Also **strmfe**, **strmfn**

strmid Return a substring from a string

Synopsis `#include <string.h>`

```
error = strmid(source,dest,pos,len);

int error;           /* -1 if pos is beyond source, else 0 */
const char *source;  /* source pointer */
char *dest;          /* destination pointer */
int pos;             /* starting position of dest in source */
int len;             /* length of substring */
```

Description The **strmid** function returns a pointer to a substring of the argument **source** beginning at character position **pos** that has a length of **len**. If **len** is greater than the length of the source offset at **pos**, then the rest of the string is copied to the destination pointed to by the **dest** argument.

The destination string is **NULL** terminated.

This function is not available if the **__STRICT_ANSI** flag has been defined.

Returns If the **pos** value is beyond the length of the **source** value, then **-1** is returned. Otherwise, **0** is returned.

Portability SAS/C

See Also **strchr**, **strrchr**, **strstr**

strncat Concatenate strings, length limited

Synopsis `#include <string.h>`

```
p = strncat(to,from,n);

char *p;           /* same as destination string pointer */
char *to;          /* destination string pointer */
const char *from;  /* source string pointer */
size_t n;          /* maximum source length */
```

Description This function concatenates the source string to the end of the destination string. No more than `n` characters are moved from the source to the destination. A `NULL` byte is placed at the end of the destination in any case.

Portability ANSI

Returns This function returns a pointer that is the same as the first argument.

Example

```
#include <stdio.h>
#include <string.h>

void main(void)
{
    char a[256],b[256];
    int n;

    while(1)
    {
        printf("\nEnter string A: ");
        if (gets(a) == NULL)
        {
            break;
        }
        printf("Enter string B: ");
        if (gets(b) == NULL)
        {
            break;
        }
    }
}
```

strncat Concatenate strings, length limited
(continued)

```
printf("Enter maximum length N: ");
if (scanf("%d",&n) == EOF)
{
    break;
}
printf("strncat(A,B,N): \"%s\"\n",
      strncat(a,b,n));
}
printf("\n");
}
```

See Also stpcpy, strcat

strncmp Compare strings, length limited*Synopsis* `#include <string.h>``x = strncmp(a,b,n);`

```

int x;                /* comparison result */
const char *a,*b;     /* strings being compared */
size_t n;

```

Description This function compares two **NULL**-terminated strings using the ASCII collating sequence. No more than **n** characters are compared.

The relative collating sequence of the strings is indicated by the sign of the return value, as follows:

Return	Meaning
negative	first string is below the second
zero	strings are equal
positive	first string is above the second

If the strings have different lengths, the shorter one is treated as if it were extended with zeroes.

Portability ANSI

Example

```

#include <stdio.h>
#include <string.h>

void result(char *name, int r)
{
    char *p;

    if (r == 0)
    {
        p = "is equal to";
    }
}

```

strncmp Compare strings, length limited
(continued)

```

        if (r < 0)
        {
            p = "is less than";
        }
        if (r > 0)
        {
            p = "is greater than";
        }
        printf("%s string A %s string B\n",name,p);
    }

void main(void)
{
    char a[256], b[256];
    int n;

    while(1)
    {
        printf("Enter string A: ");
        if (fgets(a,sizeof(a),stdin) == NULL)
        {
            break;
        }
        printf("Enter string B: ");
        if (fgets(b,sizeof(b),stdin) == NULL)
        {
            break;
        }
        printf("Enter maximum compare length: ");
        scanf("%d",&n);
        result("strncmp: ",strncmp(a,b,n));
    }
    printf("\n");
}

```

See Also `strcmp`, `strcmpi`, `stricmp`, `strnicmp`

strncpy Copy a string, length limited

Synopsis `#include <string.h>`

```
p = strncpy(to,from,n);
```

```
char *p;           /* same as destination pointer */
char *to;          /* destination pointer */
const char *from;  /* source pointer */
size_t n;          /* maximum source length */
```

Description This function copies the **NULL**-terminated source string to the destination area. The **strncpy** function always writes exactly **n** characters to the destination. If it reaches the **NULL** terminator before **n** characters are copied from the source, the destination is filled with **NULL** bytes. If the source string contains more than **n** non-**NULL** characters, the destination is not **NULL**-terminated.

Be careful when using the **strncpy** function, because it is one of the few string functions that does not produce a **NULL**-terminated string under every condition.

Portability ANSI

Returns The **strncpy** function returns a pointer that is the same as the destination pointer.

See Also **stccpy**, **stpcpy**, **strcpy**

strnicmp Compare strings, case insensitive, length limited

Synopsis `#include <string.h>`

```
x = strnicmp(a,b,n);

int x;                /* comparison result */
const char *a,*b;     /* strings being compared */
size_t n;             /* maximum comparison length */
```

Description This function compares two NULL-terminated strings using the ASCII collating sequence, but it does not distinguish between uppercase and lowercase. No more than *n* characters are compared.

The relative collating sequence of the strings is indicated by the sign of the return value, as follows:

Return	Meaning
negative	first string is below the second
zero	strings are equal
positive	first string is above the second

This function is not available if the `__STRICT_ANSI` flag has been defined.

Portability SAS/C

Returns The sign of the return value indicates the relative collating sequence of the strings, as noted above.

Example

```
#include <stdio.h>
#include <string.h>

void result(char *name, int r)
{
    char *p;

    if (r == 0)
    {
        p = "is equal to";
    }
}
```

strnicmp Compare strings, case insensitive, length limited
(continued)

```

    }
    if (r < 0)
    {
        p = "is less than";
    }
    if (r > 0)
    {
        p = "is greater than";
    }
    printf("%s string A %s string B\n",name,p);
}

void main(void)
{
    char a[256], b[256];
    size_t n;

    while(1)
    {
        printf("Enter string A: ");
        if (fgets(a,sizeof(a),stdin) == NULL)
        {
            break;
        }
        printf("Enter string B: ");
        if (fgets(b,sizeof(b),stdin) == NULL)
        {
            break;
        }
        printf("Enter maximum compare length: ");
        scanf("%d",&n);
        result("strnicmp:",strnicmp(a,b,n));
    }
    printf("\n");
}

```

See Also `strcmp`, `strcmpi`, `stricmp`, `strncmp`

strnset Set a string to a value, length limited

Synopsis `#include <string.h>`

```
p = strnset(s,c,n);

char *p; /* return pointer (same as s) */
char *s; /* string pointer */
int c;   /* value */
int n;   /* maximum string length */
```

Description This function sets all bytes of a **NULL**-terminated string to the same value, not including the terminator byte. No more than **n** bytes are set. That is, if a **NULL** byte is not found by the time **n** bytes have been set, the operation stops.

This function is not available if the `__STRICT_ANSI` flag has been defined.

Portability XENIX

Returns This function returns the original string pointer.

See Also `strset`

strpbrk Find a break character in a string

Synopsis `#include <string.h>`

```
p = strpbrk(s,b);

char *p;          /* points to break character in s */
const char *s;    /* string to be scanned */
const char *b;    /* break characters */
```

Description This function scans string **s** to find the first occurrence of a character from break string **b**.

Portability ANSI

Returns If no character from string **b** is found in string **s**, a **NULL** pointer is returned. Otherwise, **p** is a pointer to the first break character.

Example

```
/*
 * Scan for commas, periods, and blanks. Display the tail
 * of the string each time a break character is found.
 */
#include <string.h>
#include <stdio.h>

char *p, s[] = "Hello, I must be going.";

void main(void)
{
    for (p = s; (p = strpbrk(p,",. ")) != NULL; p++)
    {
        printf("%s\n",p);
    }
}
```

See Also `stpbrk`, `strspn`, `strcspn`

strrchr Find the last occurrence of a character in a string

Synopsis `#include <string.h>`

```
p = strrchr(s,c);

char *p;          /* updated string pointer */
const char *s;    /* input string pointer */
int c;            /* character to be located */
```

Description The **strrchr** function searches an input string for the last occurrence of the character **c**. The **strrchr** function is the reverse of the **strchr** function.

The input string must end with the **NULL** character (`'\0'`). A protection or addressing exception may occur if the input string is not terminated with the **NULL** character.

Returns This function returns a character pointer to the last occurrence of the search character in the input string or **NULL** if the character is not found. If the search character is the **NULL** character (`'\0'`), the pointer points to the **NULL** character at the end of the input string.

Portability ANSI

Example

```
#include <stdio.h>
#include <string.h>
#define LENGTH 80

void main(void)
{
    char line[LENGTH];
    char *last_blank;
    FILE *m;
    .
    .
    .
    /* Read a string addressed by line from */
    /* the input file associated with m.    */
    fgets(line,LENGTH,m);

    /* Find the last space character in the */
    /* string line and save its location in */
    /* last_blank.                          */
    .
    .
    .
}
```

strchr Find the last occurrence of a character in a string
(continued)

```
        last_blank = strchr(line, ' ');  
    }
```

See Also **stpchr, stpchrn, strchr**

strrev Reverse a character string

Synopsis `#include <string.h>`

```
p = strrev(s);
```

```
char *p,*s; /* string pointer */
```

Description This function reverses a character string. In other words, it rotates the string about its midpoint such that the last character is first and the first is last.

This function is not available if the `__STRICT_ANSI` flag has been defined.

Portability XENIX

Returns This function returns the same pointer that was passed to it.

strset Set a string to a value

Synopsis `#include <string.h>`

```
p = strset(s,c);
```

```
char *p; /* return pointer (same as s) */  
char *s; /* string pointer */  
int c;   /* value */
```

Description This function sets all bytes of a **NULL**-terminated string to the same value, not including the terminator byte.

This function is not available if the **__STRICT_ANSI** flag has been defined.

Portability XENIX

Returns This function returns the original string pointer.

strsfm Split the filename

Synopsis `#include <string.h>`

```
void strsfm(file,drive,path,node,ext);

const char *file;    /* file name pointer */
char *drive;         /* drive code pointer */
char *path;          /* directory path pointer */
char *node;          /* node pointer */
char *ext;           /* extension pointer */
```

Description This function splits a filename into four possible components and places them into the **drive**, **path**, **node**, and **ext** strings. If any of the arguments is **NULL**, then that component is discarded.

A complete filename is constructed as follows:

drive:path/node.ext

When the **strsfm** function splits the **file** name, it leaves the colon attached to the **drive** string, but drops the leading or trailing punctuation characters. In other words, the **path** component does not end with a slash, and the **ext** component does not begin with a period. Slashes within the **path** component are preserved.

Make sure that the **drive**, **path**, **node**, and **ext** pointers refer to areas that are large enough to hold the largest string that might be generated. This function does not check that any component lengths are exceeded, although it does copy the file string to an internal buffer of size **FMSIZE** and truncate it if it is too long. If you want to be absolutely sure that no overflows occur, make each component area **FMSIZE** bytes long.

This function is not available if the **__STRICT_ANSI** flag has been defined.

Portability SAS/C

Example

```
/* This program splits file names that it reads */
/* from stdin.                                     */

#include <stdio.h>
#include <string.h>
#include <dos.h>
```

strsf Split the filename
(continued)

```

/* After the next statement, the component strings are: */
/*  drive => ""           path => "abc/def"           */
/*  node  => "ghi"        ext  => ""                 */
/* strsf("abc/def/ghi",drive,path,node,ext);         */
/*                                                  */
/* After the next statement, the component strings are: */
/*  drive => "df0:"        path => ""                 */
/*  node  => "myfile"      ext  => "str"              */
/* strsf("df0:myfile.str",drive,path,node,ext);       */

void main(void)
{
    char input[255],drive[FNSIZE],path[FMSIZE],
        node[FNSIZE],ext[FESIZE];

    while(1)
    {
        puts("\nEnter file name...\n");
        if (gets(input) == NULL)
        {
            break;
        }
        strsf(input,drive,path,node,ext);
        printf("DRIVE:  \"%s\"\n"
              "PATH:    \"%s\"\n"
              "NODE:    \"%s\"\n"
              "EXT:     \"%s\"\n",
              drive, path, node, ext);
    }
    printf("\n");
}

```

See Also `stcgfn`, `strmfe`, `strmf`

strspn Count the number of string characters in the set

Synopsis `#include <string.h>`

```
length = strspn(s,b);

size_t length;    /* span length in bytes */
const char *s;    /* points to string being scanned */
const char *b;    /* points to character set string */
```

Description This function counts the number of characters in the input string *s* that are in the character set specified by string *b*. The scan always stops when the NULL terminator byte is reached.

Returns This function returns the number of bytes that are in the specified character set.

Portability ANSI

Example

```
#include <stdio.h>
#include <string.h>

void main(void)
{
    char s1[256],s2[256];

    while(1)
    {
        printf("\nEnter test string: ");
        if (fgets(s1,sizeof(s1),stdin) == NULL)
        {
            break;
        }
        printf("Enter span string: ");
        if (fgets(s2,sizeof(s2),stdin) == NULL)
        {
            break;
        }
        printf("strspn: %d\n",strspn(s1,s2));
    }
    printf("\n");
}
```


strspn Count the number of string characters in the set
(continued)

See Also **stcisc, stciscn, strcspn**

strsrt Sort a string pointer list

Synopsis `#include <string.h>`

```
void strsrt(s,n);

char *s[]; /* string pointer list */
int n;     /* number of pointers in list */
```

Description This function performs a simple bubble sort of the string pointers in the specified list. It is particularly useful in conjunction with the `getfnl` and `strbpl` functions. For large lists, you usually get better performance using the `tqsort` function.

This function is not available if the `__STRICT_ANSI` flag has been defined.

Portability SAS/C

Example

```
/*
 * This program constructs an array of pointers to all normal
 * files in the current directory that have an extension of
 * .c. Then the array is sorted into ASCII order.
 */
#include <stdio.h>
#include <stdlib.h>

extern long _OSERR;

void main(void)
{
    char names[3000],*pointers[300];
    int count, i;

    count = getfnl("#?.c",names,sizeof(names),0);

    if (count > 0)
    {
        if (strbpl(pointers,300,names) != count)
        {
            fprintf(stderr,"Too many file names\n");
            exit(EXIT_FAILURE);
        }
        strsrt(pointers,count);
    }
```

strsrt Sort a string pointer list
(continued)

```

        printf("Names Found:\n");
        for (i=0; i<count; i++)
        {
            printf("%s\n", pointers[i]);
        }
    }
    else
    {
        if (_OSERR)
        {
            poserr("FILES");
        }
        else
        {
            fprintf(stderr, "Too many files\n");
        }
        exit(EXIT_FAILURE);
    }
}

```

See Also `getfnl`, `strbpl`, `tqsort`

strstr Find a substring inside of a string

Synopsis `#include <string.h>`

```
p = strstr(s1, s2);

char *p;          /* pointer to substring in string.*/
const char *s1;   /* string to search in.          */
const char *s2;   /* substring to search for.      */
```

Description This function searches for a substring inside of a string and returns its position in the string.

Portability ANSI

Returns This function returns a pointer to substring `s2` within string `s1`. If it cannot find the substring, it returns a `NULL` pointer.

Example

```
/*
 * look for something in 'McDonald's farm'
 */
#include <stdio.h>
#include <string.h>

void main(void)
{
    char *p;
    p = strstr("Old McDonald had a farm, EIEIO", "arm");
    if (p == NULL)
    {
        printf("This is odd, I thought it "
               "was in there\n");
    }
    else
    {
        /*
         * This should print out:
         * Found it, it goes: arm, EIEIO
         */
        printf("Found it, it goes: %s\n", p);
    }
}
```

strstr Find a substring inside of a string
(continued)

See Also `strchr`, `strmid`, `strrchr`

strtod Convert a string to a double-precision floating-point number

Synopsis `#include <stdlib.h>`

```
d = strtod(nptr, endptr);

double d;                /* converted number      */
const char *nptr;         /* string to convert    */
char **endptr;            /* pointer to end of string */
```

Description This function converts a character string into a double-precision floating-point number. If the `endptr` pointer is not `NULL`, the function places a pointer to the following character into it. Leading white space is ignored. The `strtod` function consults the current locale information to determine the normal representation of a floating-point number for the current locale.

Portability ANSI

Returns This function returns the converted double-precision floating-point number. If it is unable to convert the input source string, it returns a value of 0. For values too large or too small to fit in a double-precision floating-point number, it returns a value that is plus or minus `HUGE_VAL` and sets the external integer `errno` to `ERANGE`. In the case of an underflow, it returns 0, and the external integer `errno` is set to `ERANGE`.

Example

```
/*
 * Read a number and convert it to a double.
 */
#include <stdio.h>
#include <stdlib.h>

char buf[80];
double d;
char *p;

void main(void)
{
    printf("Enter a BIG number:\n");
    fflush(stdout);
    fgets(buf, sizeof(buf), stdin);
    d = strtod(buf, &p);
```

strtod Convert a string to a double-precision floating-point number
(continued)

```
        printf("We got %lf, with a remaining string "  
              "of '%s'\n", d, p);  
    }
```

See Also `scanf`

strtok Get a token

Synopsis `#include <string.h>`

```
t = strtok(s,b);

char *t;          /* token pointer */
char *s;          /* input string pointer or NULL */
const char *b;    /* break character string pointer */
```

Description This function treats the input string as a series of one or more tokens separated by one or more characters from the break string. By making a sequence of calls to the **strtok** function, you can obtain the tokens in left-to-right order. To get the first (leftmost) token, supply a non-**NULL** pointer for the **s** argument. To get the next tokens, call the function repeatedly with a **NULL** pointer for the **s** argument, until you get a **NULL** return pointer to indicate that there are no more tokens. The break string can be changed from one call to another.

Each time it is entered, the **strtok** function takes the following steps:

1. If the input string is **NULL**, the **strtok** function gets the string pointer that was used on the preceding call. Otherwise, it uses the new input string pointer.
2. The **strtok** function scans forward through the string to the next nonbreak character. If it is a **NULL** byte, the **strtok** function returns a value of **NULL** to indicate that there are no more tokens.
3. The **strtok** function scans forward through the string to the next break character or the **NULL** terminator byte. In the former case, it writes a **NULL** byte into the string to terminate the token, and then scans forward until the next nonbreak character is found. In either case, it saves the final value of the string pointer for the next call and returns the token pointer.

The input string gets changed as the scan progresses. Specifically, a **NULL** byte is written at the end of each token.

Portability ANSI

Returns A **NULL** pointer is returned when there are no more tokens.

strtok Get a token
(continued)

Example

```
/*
 * This example breaks out words that are separated
 * by blanks or commas. The token pointer takes on
 * the following values as the program loops:
 *
 *   LOOP   TOKEN
 *   1      "first"
 *   2      "second"
 *   3      "third"
 *   4      "fourth"
 *   5      NULL
 */
#include <string.h>
#include <stdio.h>

char test[] = "first, second third, fourth";

void main(void)
{
    char *token;

    token = strtok(test, ", ");
    while(token != NULL)
    {
        printf("%s\n", token);
        token = strtok(NULL, ", ");
    }
}
```

See Also `stptok`, `strcspn`, `strspn`

strtol Convert a string to a long integer

Synopsis `#include <stdlib.h>`

```

r = strtol(p,np,base);

long int r;           /* result */
const char *p;        /* string pointer */
char **np;            /* returns updated string pointer */
int base;             /* conversion base */

```

Description This function converts an ASCII string into a long integer according to the specified base, which can range from 2 to 36. Valid digit characters are 0 to 9, a to z, and A to Z. The highest allowable character is determined by the base. If the base is 0, then the string is analyzed to see if it is base 8, base 10, or base 16. If the string begins with 0x or 0X, it is base 16; if the string begins with 0, it is base 8; if the string begins with a digit from 1 to 9, it is base 10.

If the pointer `np` is not `NULL`, it specifies an area into which the updated string pointer is placed. That is, the pointer `*np` points to the first character in string `p` that is not a valid digit.

The function skips blanks before starting the scan.

Portability ANSI

Returns This function returns the converted value. If it cannot do the conversion, it returns 0. If the converted value is too large or small to fit in a long integer, it returns `LONG_MAX` or `LONG_MIN` and sets the external integer `errno` to `ERANGE`.

Example

```

/*
 * Input a number from the user
 */
#include <stdio.h>
#include <stdlib.h>

long age;
char buf[256];
char *p;

```

strtol Convert a string to a long integer
(continued)

```
void main(void)
{
    printf("Please enter your age, followed by "
           "your name, on the same line\n");
    fflush(stdout);
    gets(buf);

    age = strtol(buf, &p, 10); /* We only use decimal here */
    printf("%s is %ld years old\n", p, age);
}
```

See Also `strtoul`

strtoul Convert a string to an unsigned long integer

Synopsis `#include <stdlib.h>`

```

r = strtoul(p,np,base);

unsigned long int r;    /* result */
const char *p;         /* string pointer */
char **np;             /* returns updated string pointer */
int base;              /* conversion base */

```

Description This function converts an ASCII string into an unsigned long integer according to the specified base, which can range from 2 to 36. Valid digit characters are 0 to 9, a to z, and A to Z. The highest allowable character is determined by the base. If the base is 0, then the string is analyzed to see if it is base 8, base 10, or base 16. If the string begins with 0x or 0X, it is base 16; if the string begins with 0, it is base 8; if the string begins with a digit from 1 to 9, it is base 10.

If the pointer `np` is not `NULL`, it specifies an area into which the updated string pointer is placed. That is, the pointer `*np` points to the first character in string `p` that is not a valid digit.

The function skips blanks before starting the scan.

Portability ANSI

Returns This function returns the converted value. If it cannot do the conversion, it returns 0. If the converted value is too large to fit in an unsigned long integer, it returns `ULONG_MAX` and sets the external integer `errno` to `ERANGE`.

Example

```

/*
 * Input a number from the user
 */
#include <stdio.h>
#include <stdlib.h>

unsigned long age;
char buf[256];
char *p;

```

strtoul Convert a string to an unsigned long integer
(continued)

```
void main(void)
{
    printf("Please enter your age, followed by "
           "your name, on the same line\n");
    fflush(stdout);
    gets(buf);

    age = strtoul(buf, &p, 10); /* We only use decimal here */
    printf("%s is %ld years old\n", p, age);
}
```

See Also `strtol`

strupr Convert a string to uppercase

Synopsis `#include <string.h>`

```
p = strupr(s);
```

```
char *p; /* return pointer (same as s) */  
char *s; /* string pointer */
```

Description This function converts all alphabetic characters in the specified **NULL**-terminated string to uppercase using the 7-bit ASCII collating sequence.
This function is not available if the `__STRICT_ANSI` flag has been defined.

Portability XENIX

Returns This function returns the original string pointer.

See Also `stricmp, strlwr, tolower, toupper`

strxfrm Transform a string

Synopsis `#include <string.h>`

```
len = strxfrm(s1, s2, n);
```

```
size_t len;           /* length of transformed string    */
char *s1;             /* pointer to transformed string    */
const char *s2;       /* string to transform              */
size_t n;             /* max characters in transformed string */
```

Description This function transforms strings such that the `strcmp` function will give the same result when applied to two transformed strings as the `strcoll` function would give to the same strings before they were transformed.

Because of the limitations of the ANSI specifications, this function does not completely handle the German collating rules.

Portability ANSI

Returns This function returns the length of the transformed string (excluding the NULL termination character). If a value is greater than the argument `n`, then the transformation could not fit in the output array, and the contents of the output array are indeterminate.

Example `/* Transform a string for comparison */`

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

char *transform(char *string)
{
    int len;
    char *p;

    /* Don't forget the NULL */
    len = strxfrm(string, NULL, 0)+1;
    if ((p = malloc(len)) != NULL)
    {
        strxfrm(string, p, len);
    }
    return(p);
}
```

strxfrm Transform a string
(continued)

```
void main(int argc, char *argv[])
{
    int i;

    for (i=1; i<argc; i++)
    {
        printf("%s ==> %s\n",
               argv[i], transform(argv[i]));
    }
}
```

See Also `setlocale`, `strcoll`

stspfp Parse a file path

Synopsis `#include <string.h>`

```
error = stspfp(path,nx);

int error;           /* -1 for error, 0 for success */
const char *path;    /* file path string */
int nx[16];          /* node index array */
```

Description This function parses a file path, which is a **NULL**-terminated string consisting of nodes separated by the **__SLASH** character. Each separator is replaced with a **NULL** byte, and the index to the first character of that node is placed into the node index array. The last entry in the array is followed by a **-1**. A leading separator in the path string is skipped.

This function is not available if the **__STRICT_ANSI** flag has been defined.

Portability SAS/C

Returns A return value of **-1** indicates that the path contains more than 15 nodes.

See Also **stcgfe**, **stcgfn**, **stcgfp**, **strsfn**

`_stub` Default routine for undefined routines

Synopsis `void _stub(void);`

Description The `_stub` function is the default routine resolved by the linker for routines not found in libraries. By default, it puts up a requester indicating that the unwritten routine has been called. It is intended to allow development and testing of a program for which some of the routines have not been written (and, of course, are not expected to be called).

For a list of the SAS/C libraries available, see Chapter 3, “Using the SAS/C Libraries.”

Portability AmigaDOS

swmem Swap two memory blocks

Synopsis `#include <string.h>`

```
void swmem(a,b,n);
```

```
void *a,*b;    /* block pointers */
unsigned n;    /* number of bytes */
```

Description This function swaps two blocks of memory. This function neither recognizes nor produces the **NULL** terminator byte usually found at the end of strings.

This function is not available if the **__STRICT_ANSI** flag has been defined.

Portability UNIX

Example

```
#include <stdio.h>
#include <string.h>

char buf[] = "String 1";
char buf2[] = "String 2";

void main(void)
{
    printf("%s %s\n",
           buf, buf2); /* Prints "String 1 String 2". */
    swmem(buf, buf2, strlen(buf));
    printf("%s %s\n",
           buf, buf2); /* Prints "String2 String 1". */
}
```

system Call the system command processor

Synopsis `#include <stdlib.h>`

```
error = system(cmd);

int error;          /* non-zero if error */
const char *cmd;    /* command string */
```

Description This function invokes the system command processor and passes the `cmd` string to it. Under AmigaDOS, this function invokes the AmigaDOS **Execute** function.

Portability ANSI

Returns If the command processor cannot be invoked, a value of `-1` is returned, and additional error information can be found in the external integers `errno` and `_OSERR`. Otherwise, the function returns the value passed back by the command processor.

Example

```
#include <stdio.h>
#include <stdlib.h>

void main(void)
{
    int x;

    x = system("dir ram:");
    if (x < 0)
    {
        printf("System command failed\n");
    }
    if (x > 0)
    {
        printf("System command error %d\n", x);
    }
}
```

See Also `errno`, `_OSERR`

tan Tangent function

Synopsis `#include <math.h>`

```
r = tan(x);
```

```
double r; /* result */  
double x; /* angle */
```

Description The **tan** function computes the tangent of an angle expressed in radians.

Portability ANSI

See Also `__matherr`

tanh Hyperbolic tangent function

Synopsis `#include <math.h>`

```
r = tanh(x);

double r;    /* result */
double x;    /* angle */
```

Description The `tanh` function computes the normal hyperbolic tangent of the argument `x`.

Portability ANSI

See Also `__matherr`

tell Get the level 1 file position

Synopsis `#include <fcntl.h>`

```
apos = tell(fh);

long apos; /* absolute file position */
int fh;    /* file handle */
```

Description The `tell` function is equivalent to:

```
apos = lseek(fh, 0L, 1);
```

The `tell` function returns a file position that can be used in a subsequent call to the `lseek` function to restore the file to the position at the time of the `tell` call.

Portability UNIX

Returns This function returns `-1L` if an error occurs, in which case the external integers `errno` and `_OSERR` contain additional error information.

See Also `errno`, `ftell`, `lseek`, `open`, `_OSERR`

telldir Get the directory position

Synopsis `#include <dir.h>`

```
loc = telldir(dfd);

long loc; /* current read position */
DIR *dfd; /* directory file descriptor */
```

Description This routine returns the current read position for the given file descriptor. This position is where the **readdir** function would obtain the next directory entry if it were called. The **loc** argument is also the value that you would pass to the **seekdir** function if you wanted to return directly to this same position.

loc values are good only for the life of the file descriptor. If you close a directory and reopen it, the **loc** values are no longer valid.

Portability UNIX

Returns This function returns the location of the next directory entry.

See Also **closedir**, **opendir**, **readdir**, **seekdir**

time Get the system time in seconds

Synopsis `#include <time.h>`

```
timeval = time(timeptr);

time_t timeval;    /* time value */
time_t *timeptr;   /* pointer to time value storage */
```

Description This function returns the current time expressed as the number of seconds since 00:00:00 Greenwich Mean Time, January 1, 1970. If the `timeptr` pointer is not `NULL`, the time value is also stored in that location.

Portability ANSI

Example

```
#include <stdio.h>
#include <time.h>

void main(void)
{
    long t;

    time(&t);
    printf("Current time is %s\n", ctime(&t));
}
```

See Also `asctime`, `ctime`, `gmtime`, `localtime`, `tzset`

__timecvt Convert a `time_t` value to an AmigaDOS `DateStamp`

Synopsis `#include <time.h>`

```
tp = __timecvt(t);
```

```
struct DateStamp *tp; /* pointer to converted DateStamp */
time_t t;             /* time value to convert */
```

Description This function converts a date and time in the format that the `time` function returns to the AmigaDOS `DateStamp` format.

Portability AmigaDOS

Returns This function returns a pointer to a `DateStamp` array. This array is considered read only and is valid only until the next call to the `__timecvt` function.

Example

```
/*
 * Convert the current time.
 */
#include <stdio.h>
#include <time.h>
#include <dos.h>

void main(void)
{
    struct DateStamp *event;

    event = __timecvt(time(NULL));

    printf("Days since 1JAN78      = %ld\n"
           "Minutes after midnight = %ld\n"
           "Ticks past the minute   = %ld\n",
           event->ds_Days, event->ds_Minute, event->ds_Tick);
}
```

See Also `mktime`

timer Get the system clock with microseconds

Synopsis `#include <time.h>`

```
x = timer(clock);
```

```
int x;  
unsigned int clock[2];
```

Description The **timer** function obtains the current setting of the system clock in the form of a two-integer array as follows:

```
clock[0] => seconds  
clock[1] => microseconds
```

This function is not available if the `__STRICT_ANSI` flag has been defined.

Portability AmigaDOS

Returns If successful, this function returns a 0. Otherwise, it returns a `-1`.

__tinymain Special version of the `__main` function

Synopsis `#include <stdlib.h>`

```
void __stdargs __tinymain(line);
```

```
char *line; /* ptr to command line that caused execution */
```

Description The `__tinymain` function is a special case of the standard C preprocessing function `__main`. It does not open the standard I/O files `stdin`, `stdout`, and `stderr`, and therefore, eliminates support for standard I/O routines from your executable code. If you use the `__tinymain` function, you cannot use standard I/O functions such as `printf` and `scanf`. Also, using the `__tinymain` function suppresses the initial input and output window normally displayed when starting up from the Workbench.

If you are not using registerized parameters, you can use the `__tinymain` function by specifying the following on the `slink` command line:

```
DEFINE ___main=___tinymain
```

If you use the `stdio` option in the `sc` command, the compiler handles this for you. Three underscores are required instead of two because when you compile your program, the compiler adds an underscore to symbol names.

Portability SAS/C

See Also `main`

tmpfile Open a temporary file stream

Synopsis `#include <stdio.h>`

```
fp = tmpfile(void);
```

```
FILE *fp;          /* pointer to temporary file stream */
```

Description This function opens a temporary file. The name of the temporary file is constructed from a combination of the first three characters of the program name, a string of characters based on the base stack pointer, and a sequential number (*nn*):

```
T:progname.stack.T.nn
```

The **tmpfile** function attempts to open files of increasing sequential numbers until it succeeds, or it has tried 99 times. If you do not have the argument **T**: assigned, the **tmpfile** function will never be able to create a temporary file.

Portability ANSI

Returns This function returns a file handle for a file that can be read or written. When this file is closed, it is automatically deleted.

Example

```
/*
 * Create a temporary file to hold a sort data set
 */
#include <stdio.h>

void sortfile(FILE *fpo, FILE *fpi)
{
    /* Insert code for "World's Greatest File Sorter" */
    /* in this routine...                               */
    return;
}

void main(void)
{
    FILE *fp, *infp, *outfp;
```

tmpfile Open a temporary file stream
(continued)

```
fp = tmpfile();
if (fp == NULL)
{
    printf("Can't create sort temp file\n");
    return;
}
sortfile(fp, infp);
sortfile(fp, outfp);
fclose(fp);
}
```

See Also `open`, `close`, `tmpnam`

tmpnam Create a temporary filename

Synopsis `#include <stdio.h>`

```
name = tmpnam(b);
```

```
char *name; /* pointer to temporary file name */  
char *b;    /* pointer to name buffer or NULL */
```

Description This function creates a unique name that can be used for a temporary file. If the **b** pointer is not **NULL**, it should point to a buffer at least **L_tmpnam** bytes long, and the function returns that pointer. If **b** is **NULL**, then the function creates a buffer and returns a pointer to it. Subsequent calls to the **tmpnam** function may modify the buffer.

Portability ANSI

Returns This function returns a pointer to a temporary filename.

See Also **tmpfile**

to..... Convert a character to ASCII, lowercase or uppercase

Synopsis `#include <ctype.h>`

```
cc = toascii(c);
cc = tolower(c);
cc = toupper(c);

int cc; /* converted character */
int c;  /* character to convert */
```

Description The `toascii` function resets all high-order bits, leaving only the lower seven. The `tolower` function tests if the argument `c` is an uppercase alphabetic character and, if so, converts it to lowercase. The `toupper` function is the reverse of the `tolower` function.

If you include the file `ctype.h` as shown above, these functions are actually defined as macros and produce inline code to perform the conversions. Without the `ctype.h` file, they are actual functions resolved in the standard library. If you want to use the function version but must include the file `ctype.h` for some other reason, use an `#undef` statement to undefine the macros after including the `ctype.h` file.

The `toascii` function is not available if the `__STRICT_ANSI` flag has been defined.

Portability

SAS/C	<code>toascii</code>
ANSI	all others

Example

```
/*
 * The following program echoes each input
 * line in upper case.
 */
#include <stdio.h>
#include <ctype.h>
```


to..... Convert a character to ASCII, lowercase or uppercase
(continued)

```
void main(void)
{
    char b[256],*p;

    while(gets(b) != NULL)
    {
        for (p = b; *p != '\0'; p++)
        {
            *p = toupper(*p);
        }
        puts(b);
    }
}
```

See Also `__ctype`, `isascii`, `islower`

tqsort Sort an array of text pointers

Synopsis `#include <stdlib.h>`

```
void tqsort(ta,n);
```

```
char *ta[];      /* pointer to text pointer array */
size_t n;        /* number of elements in array */
```

Description The **tqsort** function sorts the specified data array using the ACM 271 algorithm, more popularly known as Quicksort. During its operation, it calls on the **strcmp** comparison routine with pointers to the two array elements being compared.

The **ta** array consists of pointers to **NULL**-terminated character strings. This function rearranges the pointers so that the strings are in ascending ASCII sequence. The sort is based on the contents of the strings rather than their physical address.

This function is not available if the **__STRICT_ANSI** flag has been defined.

Portability SAS/C

See Also **dqsort**, **fqsort**, **lqsort**, **qsort**, **sqsort**

__tzset Set the time zone variables

Synopsis `#include <time.h>`

```
void __tzset(void);

/* If the _STRICT_ANSI flag has not been defined,
 * these symbols are defined in time.h:
 * extern int __daylight;
 * extern long __timezone;
 * extern char *__tzname[2];
 * extern char __tzstn[4];
 * extern char __tzdtn[4];
 * extern char *_TZ;
 */
```

Description The `__tzset` function assigns values to the time zone variables `__daylight`, `__timezone`, and `__tzname`. These variables are then used by the `localtime` function and other functions to correct from Greenwich Mean Time (GMT) to local time.

The values for these variables are obtained from the character string pointer named `_TZ`, which has the following form:

```
char *_TZ = "aaabbbccc"
```

`aaa` is the three-letter abbreviation for the local standard time zone (for example, CST), and `bbb` is a number from `-23` to `24` that specifies the number to be subtracted from GMT to obtain local standard time. Both `aaa` and `bbb` are required. `ccc` is the abbreviation for the local daylight savings time zone (for example, CDT), and it should be present only if daylight savings time is currently in effect.

Initially, the `_TZ` pointer is set to `NULL`. It should be initialized with the address of a string corresponding to the correct time zone. If `_TZ` is `NULL`, the `__tzset` function uses the default string `CDT6`.

When the `__tzset` function is called, the `__timezone` integer is loaded with the number of seconds that must be subtracted from GMT to get the local time. Next, the `__daylight` integer is loaded with `0` if the `ccc` portion of the `_TZ` pointer is absent and `1` if `ccc` is present. Then, the `aaa` and `ccc` parts are copied to `__tzstn` and `__tzdtn`, respectively, with `NULL` terminators. Finally, `__tzname[0]` and `__tzname[1]` are loaded with pointers to `__tzstn` and `__tzdtn`, respectively.

The symbols defined in the file `time.h` are not available if the `_STRICT_ANSI` flag has been defined.

— **__tzset** Set the time zone variables
(continued)

Portability XENIX

See Also **localtime**

ungetc Push input character back

Synopsis `#include <stdio.h>`

```

r = ungetc(c, fp);

int r;      /* return character or code */
int c;      /* character to be pushed back */
FILE *fp;   /* file pointer */

```

Description This function pushes a character back to the specified level 2 input file. The character need not be the same as the one that was most recently read. However, before calling the **ungetc** function, you must have read at least one character using the **fgetc** function or one of the other level 2 input functions. Also, you can only push back one character; if you call the **ungetc** function more than once between input functions, the results are undefined.

Portability ANSI

Returns Normally, the **ungetc** function returns the character that was pushed back. However, if the end-of-file has been reached or if no characters have been read yet, the value **EOF** is returned.

Example `#include <stdio.h>`

```

void main(void)
{
    int c;

    while(1)
    {
        printf("Loop 1...\n");
        while((c = getchar()) != EOF)
        {
            if (isalpha(c))
            {
                putchar(c);
            }
            else
            {
                break;
            }
        }
    }
}

```

ungetc Push input character back
(continued)

```
    }
    ungetc(c, stdin);
    printf("\n\nLoop 2...\n");
    while((c = getchar()) != EOF)
    {
        if (isalpha(c) == 0)
        {
            putchar(c);
        }
        else
        {
            break;
        }
    }
    ungetc(c, stdin);
}
printf("\n\nDone\n");
}
```

unlink Remove a file

Synopsis `#include <stdio.h>`

```
error = unlink(name);

int error;           /* non-zero if error */
const char *name;    /* file name */
```

Description This function removes the specified file from the system. The file **name** argument can include a path, but it cannot include wild card characters. That is, you can remove only one file at a time.

The **unlink** function is provided for compatibility with some versions of UNIX.

This function is not available if the `__STRICT_ANSI` flag has been defined.

Portability UNIX

Returns If a nonzero value is returned, some type of error occurred, and additional information can be found in the external integers `errno` and `OSERR`. The most common errors occur when you try to remove a file that doesn't exist, that is marked as read-only, or that is in use.

Example

```
/*
 * This program removes all files specified
 * in the argument list. It does not allow
 * wild card characters in the file names.
 */
#include <stdio.h>
#include <stdlib.h>

void main(int argc, char *argv[])
{
    int i;           /* loop counter */
    /* exit code, non-zero if any failures */
    int ret = EXIT_SUCCESS;
```

unlink Remove a file
(continued)

```
    for (i = 1; i < argc; i++)
    {
        if (unlink(argv[i]))
        {
            perror("RMV");
            ret = EXIT_FAILURE;
        }
    }
    exit(ret);
}
```

See Also `errno`, `_OSERR`, `remove`

utpack Pack UNIX time

Synopsis `#include <time.h>`

```
ut = utpack(x);

long ut;          /* packed UNIX time */
const char *x;    /* unpacked UNIX time */
```

Description This function packs the 32-bit time value that is traditionally used in UNIX systems. This value is the number of seconds since 00:00:00, January 1, 1970. The `time` function returns the system clock in this form relative to Greenwich Mean Time.

The unpacked time is a 6-byte array in the following format:

Byte	Contents
<code>x[0]</code>	year - 1970 (−128 to 127)
<code>x[1]</code>	month (1 to 12)
<code>x[2]</code>	day (1 to 31)
<code>x[3]</code>	hour (0 to 23)
<code>x[4]</code>	minute (0 to 59)
<code>x[5]</code>	second (0 to 59)

Although this array is similar to the one produced by the `getclock` function and used by the `stptime` function, the year is biased relative to 1970 instead of 1980. The year is a signed character and can be negative. A value of −3, for example, is 1967 (in other words, 1970 − 3).

This function is not available if the `__STRICT_ANSI` flag has been defined.

Portability SAS/C

Returns This function returns a packed long integer, as described above.

Example

```
/*
 * Get a file time and subtract 10 years from it.
```

utpack Pack UNIX time
(continued)

```

* No error checks.
*/
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <dos.h>

void main(int argc, char *argv[])
{
    char tt[6];
    long ft,ut;

    ft = getft(argv[1]);
    utunpk(ft, tt);
    tt[0] -= 10;
    ut = utpack(tt);
    printf("File time is: %s\n",ctime(&ut));
}

```

See Also ctime, getclk, gmtime, localtime, stpdate, time, utunpk

utunpk Unpack UNIX time

Synopsis `#include <time.h>`

```
void utunpk(ut,x);
```

```
long ut; /* packed UNIX time */
char *x; /* unpacked UNIX time */
```

Description This function unpacks the 32-bit time value that is traditionally used in UNIX systems. This value is the number of seconds since 00:00:00, January 1, 1970. The `time` function returns the system clock in this form relative to Greenwich Mean Time.

The unpacked time is a 6-byte array in the following format:

Byte	Contents
<code>x[0]</code>	year - 1970 (−128 to 127)
<code>x[1]</code>	month (1 to 12)
<code>x[2]</code>	day (1 to 31)
<code>x[3]</code>	hour (0 to 23)
<code>x[4]</code>	minute (0 to 59)
<code>x[5]</code>	second (0 to 59)

Although this array is similar to the one produced by the `getclk` function and used by the `stptime` function, the year is biased relative to 1970 instead of 1980. So, if you use the `utunpk` function followed by the `stptime` function, you must subtract 10 from `x[0]` before the `stptime` call. The year is a signed character and can be negative. A value of −3, for example, is 1967 (in other words, 1970 − 3).

This function is not available if the `_STRICT_ANSI` flag has been defined.

Portability SAS/C

Example

```
/*
 * Get a file time and subtract 10 years from it.
 * No error checks.
```

utunpk Unpack UNIX time
(continued)

```
*/
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <dos.h>

void main(int argc, char *argv[])
{
    char tt[6];
    long ft,ut;

    ft = getft(argv[1]);
    utunpk(ft, tt);
    tt[0] -= 10;
    ut = utpack(tt);
    printf("File time is: %s\n", ctime(&ut));
}
```

See Also ctime, getclk, gmtime, localtime, stpdate, time, utpack

va_arg Get an argument from a varying-length argument list

Synopsis `#include <stdarg.h>`

`(arg_type) va_arg(va_list ap, arg_type);`

Description The **va_arg** macro returns the value of the next argument in a varying-length argument list.

The first argument, **ap**, is a work area of type **va_list**, which is used by various macros defined in the file **stdarg.h**. (The **va_list** must be initialized by a previous use of the **va_start** macro, and a corresponding **va_end** should be called when processing of the arguments is finished.)

The second argument, **arg_type**, is the type of the argument that is expected. The **arg_type** argument must be written in such a form that **arg_type *** is the type of a pointer to an element of that type. For example, **char** is a valid **arg_type** because **char *** is the type of a pointer to a character. **int(*)()** is not a valid second argument to the **va_arg** macro because **int(*)()*** is not a valid type. (You can use **typedef** declarations to create usable synonyms of this sort for any type.)

The results of the **va_arg** macro are unpredictable if the argument values are not appropriate.

In certain cases, arguments are converted when they are passed to another type. For instance, **char** and **short** arguments are converted to **int**, **float** to **double**, and array to pointer. When parameters of this sort are expected, the **va_arg** macro must be issued with the type after conversion. For example, **va_arg(ap, float)** may fail to access a **float** argument value correctly, so **va_arg(ap, double)** should be used.

Note: There is no way to test whether a particular argument is the last one in the list. Attempting to access arguments after the last one in the list produces unpredictable results.

Portability ANSI

Returns The **va_arg** macro returns the value of the next argument in the list. The type is always the same as the second argument to the **va_arg** macro.

va_arg Get an argument from a varying-length argument list
(continued)

Example

```

/*
 * This example shows a function named concat,
 * which can be used to concatenate any number
 * of strings. A simple call is concat(3,a,b,c).
 * This should have the same effect as
 *
 *      strcat (a,b);
 *      strcat (a,c);
 *
 * The first argument is the total number of strings.
 */

#include <stdarg.h>
#include <string.h>
#include <stdio.h>

void concat (int count, ...);

void main(void)
{
    char str[20] = "abcd";

    concat(4,str,"efgh","ijkl","mnop");

    printf("The concatenated string = %s\n",str);
}

void concat (int count, ...)
{
    va_list ap;
    char *target, *source;

    if (count <=1)
        return;

    va_start(ap, count);

```

va_arg Get an argument from a varying-length argument list
(continued)

```

        /* Get target string */
        target = va_arg (ap, char *);

        /* Point to string end */
        target += strlen(target);

        while (--count > 0)
        {
            /* Get next source string */
            source = va_arg(ap, char *);

            /* Copy chars to target */
            while (*source)
                *target++ = *source++;
        }

        /* Add final null */
        *target = '\0';

        /* End arg list processing */
        return;
    }

```

See Also `va_end`, `va_start`

va_end End varying-length argument list processing

Synopsis `#include <stdarg.h>`

```
void va_end(va_list ap);
```

Description The **va_end** macro completes processing of a varying-length argument list. The argument **ap** is a work area of type **va_list**, which is used by various macros defined in the file **stdarg.h**.

After the **va_end** macro is called, the **va_start** macro must be called again before the **va_arg** macro can be used.

In this implementation, use of the **va_end** macro in varying-length argument list processing is not required. However, in other implementations, failure to issue the **va_end** macro may cause program failures on return from the function that issued the **va_start** macro.

Portability ANSI

Example See the example for the **va_arg** macro.

See Also **va_arg**, **va_start**

va_start Begin varying-length argument list processing

Synopsis `#include <stdarg.h>`

```
void va_start(va_list ap, arg_name);
```

Description The **va_start** macro initializes processing of a varying-length argument list. The first argument, **ap**, is a work area of type **va_list**, which is used by various macros defined in the file **stdarg.h**. The second argument, **arg_name**, is the name of the parameter to the calling function after which the varying part of the parameter list begins (the parameter immediately before the `,...`).

The results of the **va_start** macro are unpredictable if the argument values are not appropriate.

Portability ANSI

Example See the example for the **va_arg** macro.

See Also **va_arg**, **va_end**

vfprintf Formatted write of a varying-length argument list to a file

Synopsis `#include <stdarg.h>`
`#include <stdio.h>`

```
n = vfprintf(fp, ctl, args);
```

```
int n;           /* number of characters written      */
                /* or -1 for error                      */
FILE *fp;        /* file to be written to                  */
const char *ctl; /* control string specifying formatting */
va_list args;    /* items to be formatted                 */
```

Description This function is identical in capabilities to the `fprintf` function, except that the argument list is passed as a `va_list` instead of on the stack. The argument list `args` must be initialized by the caller with a `va_start` macro (and any preceding `va_arg` macros that it wants to call). When terminated, it is the responsibility of the caller to call the `va_end` macro on the argument list.

Portability ANSI

Returns This function returns the number of characters written or, in the case of an error, a `-1`.

Example

```
#include <stdio.h>
#include <stdarg.h>

/* My own error function for a given error number */
void myerr(FILE *fp, int ernum, char *string, ...)
{
    va_list arglist;

    va_start(arglist, string);

    fprintf(fp, "ERR-%d: \n", ernum);
    vfprintf(fp, string, arglist);

    va_end(arglist);
}
```

fprintf Formatted write of a varying-length argument list to a file
(continued)

```
void main(void)
{
    myerr(stderr, 205, __sys_errlist[205]);
}
```

See Also **fprintf**

vprintf Formatted write of a varying-length argument list to standard output

Synopsis `#include <stdarg.h>`
`#include <stdio.h>`

```
x = vprintf(ctl, args);

int x;           /* number of characters written      */
                /* or -1 for error                                */
const char *ctl; /* control string specifying formatting */
va_list args;    /* items to be formatted                        */
```

Description This function is identical in capabilities to the `printf` function, except that the argument list is passed as a `va_list` instead of on the stack. The argument list `args` must be initialized by the caller with a `va_start` macro (and any preceding `va_arg` macros that it wants to call). When terminated, it is the responsibility of the caller to call the `va_end` macro on the argument list.

Portability ANSI

Returns This function returns the number of characters written or, in the case of an error, a `-1`.

Example

```
#include <stdio.h>
#include <stdarg.h>

/*
 * My own error function for a given error number
 */
void myerr(int errno, char *string, ...)
{
    va_list arglist;

    va_start(arglist, string);

    printf("ERR-%d: ", errno);
    vprintf(string, arglist);

    va_end(arglist);
}
```

vprintf Formatted write of a varying-length argument list to standard output
(continued)

```
void main(void)
{
    myerr(205, _sys_errlist[205]);
}
```

See Also `printf`

vsprintf Formatted write of a varying-length argument list to a string

Synopsis `#include <stdarg.h>`
`#include <stdio.h>`

```
x = vsprintf(buf, ctl, args);
```

```
int x;                /* number of characters placed in the */
                    /* output buffer */
char *buf;            /* String for resulting image */
const char *ctl;      /* control string specifying formatting */
va_list args;         /* items to be formatted */
```

Description This function is identical in capabilities to the `sprintf` function, except that the argument list is passed as a `va_list` instead of on the stack. The argument list `args` must be initialized by the caller with a `va_start` macro (and any preceding `va_arg` macros that it wants to call). When terminated, it is the responsibility of the caller to call the `va_end` macro on the argument list.

Portability ANSI

Returns This function returns the number of characters placed in the output buffer (excluding the terminating `NULL` byte).

Example

```
#include <stdio.h>
#include <stdarg.h>

/*
 * Format an arbitrary message into a buffer
 */
void getmsg(char *buffer, char *string, ...)
{
    va_list arglist;

    va_start(arglist, string);

    vsprintf(buffer, string, arglist);

    va_end(arglist);
}
```

vsprintf Formatted write of a varying-length argument list to a string
(continued)

```
void main(void)
{
    char buf[256];

    getmsg(buf, "Formatted with %d argument.\n", 1);
    printf(buf);
}
```

See Also **sprintf**

wait, waitm Wait for single or multiple child processes to complete

Synopsis `#include <dos.h>`

```
cc = wait(procid);
complist = waitm(proclist);

int cc;                                /* child's completion code */
struct ProcID *procid;                 /* pointer to process ID structure */
struct ProcID *complist;               /* pointer to linked list of
/*      completed process IDs      */
struct ProcID **proclist;              /* address of pointer to linked
/*      list of process IDs        */
```

Description After a process creates a child with the `fork` and `forkv` functions, the parent continues to execute until it calls either the `wait` or `waitm` function; that is, the parent and child are multiprogrammed. When the child process completes, the parent process can get its completion code with the `wait` or `waitm` function.

See the description of the `forkl` and `forkv` functions for more information.

Portability SAS/C

Returns This function returns the integer completion code of the child process.

Example See the example under the `forkl` and `forkv` functions.

See Also `exit`, `forkl`, `forkv`, AmigaDOS functions `LoadSeg` and `CreateProc` (*The AmigaDOS Manual, 3rd Edition*)

wcstombs Convert a wide-character string to a multibyte string

Synopsis `#include <stdlib.h>`

```
length = wcstombs(s, pwcs, n);
```

```
size_t length;           /* length or state information */
char *s;                 /* pointer to characters */
const wchar_t *pwcs;     /* pointer to wide-character string */
size_t n;                /* maximum number of characters to look at *
```

Description This function converts a **NULL**-terminated wide-character string to a **NULL**-terminated multibyte string. The **wcstombs** function for the current locale is passed the input parameters, and the result is returned.

Portability ANSI

See Also **mblen**, **wctomb**

wctomb Map a wide character to a multibyte character

Synopsis `#include <stdlib.h>`

```
length = wctomb(s, wc);

int length;    /* length or state information */
char *s;       /* pointer to characters or NULL */
wchar_t wc;    /* wide character */
```

Description This function converts a wide character to a multibyte character sequence. The **wctomb** function for the current locale is passed the input parameters, and the result is returned.

Portability ANSI

See Also `mblen, wcstombs`

write Write to a level 1 file

Synopsis `#include <fcntl.h>`

```
count = write(fh,buffer,length);

int count;           /* actual bytes read or written */
int fh;              /* file handle */
void *buffer;        /* data buffer */
unsigned int length; /* number of bytes to read or write */
```

Description This function writes a level 1 file whose handle was returned by the `creat` or `open` function. Under normal circumstances, the value returned should match the buffer length. If this value is `-1` or greater than the requested length, then some type of error occurred, and you should consult the external integers `errno` and `_OSERR`. If the actual length is less than the requested length when reading, this usually means that the file is exhausted. Similarly, if the actual length is less than the requested length for a write operation, this usually means that the device has no more space available. In both of these cases, it is still a good idea to check the external integers `errno` and `_OSERR` just in case some malfunction caused the short count.

Level 1 files are automatically closed by the `exit` function, which is usually called for you when the program terminates.

Portability UNIX

Returns If the operation is successful, this function returns the actual number of bytes transferred. Otherwise, it returns `-1` and places error information in the external integers `errno` and `_OSERR`.

See Also `errno`, `fwrite`, `open`, `_OSERR`, `read`,

__XCEXIT Terminate the program

Synopsis `#include <stdlib.h>`

`void __XCEXIT(lcode);`

`long lcode;`

Description This function terminates execution of the current program and returns control to the parent program. It does not close any files, but does call the standard termination routines.

 The parameter `code` is a value from 0 to 255 that gets passed back to the parent. By convention, a value of 0 indicates success. Normally, your program should call the `exit` function unless it knows that no level 1 or 2 files are open, and you are not compiling with the `nostdio` option. If you are compiling with `nostdio`, your program should call `__exit`.

Portability SAS/C

See Also `exit`, `__exit`

Index

A

abort function 103
 See also program control functions
 abs function 104
 See also mathematical macros and functions
 access function 105–106
 See also chmod function
 acos function 107
 See also cos function
 allocating memory
 See memory allocation functions
 amiga.lib 5–6, 13
 AmigaDOS JOIN command
 See JOIN command
 Append mode 203
 arccosine function
 See acos function
 arcsine function
 See asin function
 arctangent functions
 See also mathematical macros and functions
 argopt function 108–110
 See also main function
 See also system interface functions
 asctime function 111–112
 See also time functions
 asin function 113
 See also mathematical macros and functions
 — asm keyword 33
 assert macro 114–115
 astcsma function 116–117
 See also string functions

asterisk (*)
 fscanf function 231
 representing comments in function description files 29
 atan function 118
 See also mathematical macros and functions
 atan2 function 119
 See also mathematical macros and functions
 atexit function 120
 See also exit function
 atof function 121–122
 See also conversion functions
 atoi function 123
 See also conversion functions
 atol function 124
 See also conversion functions
 autodocs, ordering 1

B

— _BackgroundIO global variable 42
 See also _Backstdout global variable
 _Backstdout global variable 43
 See also — _BackgroundIO global variable
 base=abs compiler option 12, 15
 bldmem function 125
 See also memory allocation functions
 block I/O functions
 _dread 166
 _dwrite 168
 fread 225
 fwrite 239
 list of functions 96

block I/O functions (*continued*)

read 355

write 562

brackets ([]), fprintf function 214

bsearch function 126

See also qsort function

—_buffsize function 44

See also file control functions

built-in functions 20

abs 104

calls to printf function 20

disabling 20

—_emit 171

geta4 241

getreg 262–263

list of functions 20, 102

max 302

memcmp 309

memcpy 310

memset 313

min 314

putreg 345–346

strcmp 466–468

strcpy 471

strlen 481

C

calloc function 127–128

See also memory allocation
functions

ceil function 129

See also floor function

character I/O functions

fgetc 188

fgetchar 189

fputc 221

fputchar 222

getc 244

getch 246

getchar 247

is..... test functions 271–272

list of functions 96

putc 341

putchar 342

ungetc 540–541

character type macros and
functions

is..... test functions 271–272

list of functions 85–86

toascii, tolower, toupper
functions 535–536

chdir function 130

See also directory functions

chgclk function 131

See also getclk function

child process, creating

See fork1, forkv functions

Chk_Abort, chkabort functions
132See also program control
functions

chkml function 133

See also memory allocation
functions

chkufb function 134

See also —_nufbs

See also —_ufbs array

chmod function 135–136

See also access function

clearerr function 137

See also error handling
functions

clock function 138–139

See also system interface
functions

See also time functions

close function 140

See also file control functions

closedir function 141

See also directory functions

CloseLibrary function 37

clrerr function 142

See also error handling
functions

See also fopen function

comma (,)

purpose in function description
files 30

- Commodore shared libraries
 - See shared libraries
- comparing
 - memory 309
 - strings 466
- compiler options
 - base=abs 12, 15
 - gst 21
 - libcode 37
 - makegst 21
 - math=ffp 13
 - math=ieee 13
 - math=standard 13
 - math=68881 13
 - options required to compile and
 - link with each library 13–15
 - parms=register 12
 - shortint 12, 15
- compiling
 - compiling and linking with one
 - command 13–15
 - linking with custom linkable
 - library 32–33
 - options required to compile and
 - link with each library 14–15
 - sc command 6, 11–12
- compressing system header files 19–20
- concatenating strings 464
- conversion functions
 - atof 121–122
 - atoi 123
 - atol 124
 - ctime 147–148
 - ecvt 169–170
 - fcvt 182–183
 - gcvt 240
 - list of functions 88–89
 - mbstowcs 304
 - mktime 316–317
 - std_i 404–405
 - std_l 406–407
 - stch_i 413–414
 - stch_l 415–416
 - stci_d 417–418
 - stci_h 419–420
 - stci_o 421–422
 - stcl_d 427–428
 - stcl_h 429–430
 - stcl_o 431–432
 - stco_i 434–435
 - stco_l 436–437
 - stcu_d 445–446
 - stcul_d 447–448
 - stptime 454–455
 - stptime 458–459
 - strlwr 482
 - strtod 509–510
 - strtol 513–514
 - strtoul 515–516
 - strupr 517
 - __timecv 529
 - toascii 535–536
 - tolower 535–536
 - toupper 535–536
 - utpack 544–545
 - utunpk 546–547
 - wcstombs 560
- copying
 - memory 306
 - strings 403
- cos function 143
 - See also mathematical macros and functions
- cosh function 144
 - See also mathematical macros and functions
- cot function 145
 - See also mathematical macros and functions
- creat function 146
 - See also file control functions
 - See also file management functions
- Create mode 202

creating libraries

- See linkable libraries, custom

- See shared libraries, custom

ctime function 147–148

- See also time functions

—_ctype array 45–46

- See also is..... test functions

custom libraries

- See linkable libraries, custom

- See shared libraries, custom

_CXFERR function 149

- See also _FPERR

- See also —_matherr function

_CXOVF function 150

— CXxnn errors 16

D

data sections 15

datecmp function 151

dates

- comparing 151

- converting 529

—_daytbl array 47–48

- See also —_monttbl array

- See also —_months array

_dclose function 152

- See also block I/O functions

- See also file control functions

- See also file positioning functions

_dcreat function 153

- See also block I/O functions

- See also file control functions

- See also file positioning functions

_dcreatx function 154

- See also block I/O functions

- See also file control functions

- See also file positioning functions

#define statement

- built-in functions called by default 20

- required before referencing

- library base 7

defining library bases

- See library bases

.device library files

- initializing library base 8

- library bases 9

dfind function 155–156

- See also directory functions

diagnostic control functions

- assert 114–115

- except 173–174

- list of functions 94

- _matherr 300–301

- perror 335

- poserr 336

- strerror 474

difftime function 157–158

- See also time functions

directory functions

- chdir 130

- closedir 141

- dfind 156–157

- dnext 161–162

- findpath 196–197

- getcd 245

- getcwd 249

- getfnl 255–257

- getpath 261

- list of functions 100–101

- mkdir 315

- opendir 332–333

- readdir 356

- rewinddir 366

- rmdir 369

- seekdir 374

- telldir 527

div function 159–160

- See also ldiv function

dnext function 161–162

- See also directory functions

_dopen function 163

- See also block I/O functions

- See also file control functions

See also file positioning functions

DOSBase 49

dqsort function 164

See also sorting functions

drand48 function 165

See also random-number generation functions

_dread function 166

See also block I/O functions

See also file control functions

See also file positioning functions

_dseek function 167

See also block I/O functions

See also file control functions

See also file positioning functions

_dwrite function 168

See also block I/O functions

See also file control functions

See also file positioning functions

E

ecvt function 169–170

See also fcvt function

__emit function 171

See also getreg function

See also putreg function

environment variables

reading 252

setting 343

erand48 function 172

See also random-number generation functions

errno 50–52

See also diagnostic control functions

See also _FPERR

error data names and functions

errno 50–52

_FPERR 55–56

__matherr function 300–301

_OSERR 70–73

__sys_errlist 78

__sys_nerr 79

error handling functions

clearerr 137

clrerr 142

feof 185

ferror 186

list of functions 97

errors

See also diagnostic control functions

internal library (__CXnn) 16

returned by AmigaDOS 70

returned by math functions 300

returned by other library functions 50

except function 173–174

See also diagnostic control functions

exit function 175–176

See also program control functions

__exit function 177

See also exit function

exit traps, setting 120

exp function 178

See also log function

F

fabs function 179

See also abs function

far keyword 15

fclose function 180

See also file control functions

fcloseall function 181

See also file control functions

fcvt function 182–183

See also ecvt function

.fd file 29

fdopen function 184

See also file control functions

- fd2pragma utility
 - creating function description (.fd)
 - file 29–30
 - creating linkable libraries 31
 - entry points for functions 30
 - instructions 29–30
 - running 30
- feof function 185
 - See also error handling functions
- ferror function 186
 - See also error handling functions
- fflush function 187
 - See also file control functions
- fgetc function 188
 - See also character I/O functions
 - See also fopen function
 - See also standard I/O functions
- fgetchar function 189
 - See also character I/O functions
 - See also fopen function
 - See also standard I/O functions
- fgetpos function 190–192
 - See also file positioning functions
- fgets function 193–194
 - See also block I/O functions
 - See also character I/O functions
 - See also error handling functions
 - See also file control functions
 - See also string I/O functions
- file control functions
 - close 140
 - creat 146
 - _dclose 152
 - _dcreat 153
 - _dcreatx 154
 - _dopen 163
 - fclose 180
 - fcloseall 181
 - fdopen 184
 - fflush 187
 - fileno 195
 - flushall 199
 - fmode 201
 - fopen 202–206
 - freopen 229
 - iomode 270
 - list of functions 97–98
 - open 330–331
 - setbuf 375
 - setnbf 381
 - setvbuf 382–383
 - tmpfile 532–533
 - tmpnam 534
- file management functions
 - access 105–106
 - chkufb 134
 - chmod 135–136
 - fileno 195
 - fmod 200
 - fstat 236–237
 - getfa 254
 - getft 258
 - iomode 270
 - list of functions 99
 - lstat 291–292
 - remove 359–360
 - rename 361–363
 - setbuf 375
 - setnbf 381
 - setvbuf 382–383
 - stat 399–400
 - stcgfe 408–409
 - stcgfn 410
 - stcgfp 411–412
 - strmfe 483
 - strmfen 484–485
 - strmfep 486
 - strsfen 501–502
 - stspfp 520
 - unlink 542–543

- file modes 202–203
- file positioning functions
 - _dseek 167
 - fgetpos 190–192
 - fseek 234
 - fsetpos 235
 - ftell 238
 - list of functions 98
 - lseek 289–290
 - rewind 365
 - tell 526
- fileno function 195
 - See also file control functions
- findpath function 196–197
 - See also getpath function
- floor function 198
 - See also ceil function
- flushall function 199
 - See also file control functions
- _fmask 53
 - See also file control functions
- fmod function 200
 - See also modf function
- _fmode 54
 - See also fopen function
- fmode function 201
 - See also file control functions
- fopen function 202–206
 - See also block I/O functions
 - See also character I/O functions
 - See also file control functions
 - See also string I/O functions
- FORKENV structure 208
- forkl, forkv functions 207–213
 - See also program control functions
 - See also structures
- formatted I/O functions
 - fprintf 214–220
 - fscanf 231–233
 - list of functions 97
 - printf 226–338
 - scanf 372
 - sprintf 389–390
 - sscanf 396
 - vfprintf 553–554
 - vprintf 555–556
 - vsprintf 557–558
- FPERR 55–56
 - See also errno
- fprintf function 214–220
 - See also formatted I/O functions
- fputc function 221
 - See also character I/O functions
 - See also fopen function
- fputchar function 222
 - See also character I/O functions
 - See also fopen function
- fputs function 223
 - See also ferror function
 - See also fopen function
 - See also fputc function
 - See also puts function
- fqsort function 224
 - See also sorting functions
- fread function 225
 - See also block I/O functions
 - See also character I/O functions
 - See also error handling functions
 - See also file control functions
 - See also fseek function
- free function 226–228
 - See also memory allocation functions
- freopen function 229
 - See also file control functions
- frexp function 230
 - See also mathematical macros and functions
- fscanf function 231–233
 - See also formatted I/O functions

fseek function 234
 See also file positioning functions
 See also fopen function
 fsetpos function 235
 See also file positioning functions
 fstat function 236–237
 See also chmod function
 ftell function 238
 See also file positioning functions
 See also fopen function
 function description file (.fd), creating 29
 functions
 See library functions
 fwrite function 239
 See also block I/O functions
 See also character I/O functions
 See also error handling functions
 See also file control functions
 See also fseek function

G

gcvt function 240
 See also conversion functions
 general utility macros and functions
 —_emit 171
 geta4 241
 getreg 262–263
 isatty 273
 list of functions 91
 offsetof 323–325
 ovlyMgr 334
 putreg 345–346
 generating random numbers
 See random-number generation functions
 getasn function 242–243
 See also system interface functions

geta4 function 241
 getc function 244
 See also character I/O functions
 See also fopen function
 getcd function 245
 See also getcwd function
 getch function 246
 See also character I/O functions
 See also fopen function
 getchar function 247
 See also character I/O functions
 See also fopen function
 getclk function 248
 See also chgclk function
 getcwd function 249
 See also getcd function
 getdfs function 250–251
 getenv function 252–253
 See also putenv function
 getfa function 254
 getfnl function 255–257
 See also directory functions
 See also strbpl function
 See also strsr function
 getft function 258
 See also ctime function
 getmem function 259
 See also memory allocation functions
 getml function 260
 See also memory allocation functions
 getpath function 261
 See also findpath function
 getreg function 262–263
 See also —_emit function
 See also putreg function
 gets function 264–265
 See also character I/O functions

See also error handling functions

See also fopen function

GfxBase 57

global data section in shared libraries 37

global symbol tables (GSTs) 21–23

- creating 21–22
- definition 21
- header file restrictions 22–23

global symbols defined by slink 38

global variables

- __BackgroundIO 42
- __Backstdout 43
- __priority 74
- __procname 75
- __stack 77
- using for library base 7

gmtime function 266–267

See also time functions

gst compiler option 21

GSTs

See global symbol tables (GSTs)

H

halloc function 268

See also memory allocation functions

header files

See system header files

I

I/O functions

- See block I/O functions
- See character I/O functions
- See error handling functions
- See file control functions
- See file positioning functions
- See formatted I/O functions
- See standard I/O functions
- See string I/O functions

iabs function 269

See also abs function

include files

- See system header files

initializing library bases 7–8

internal library errors 16

IntuitionBase 58

iomode function 270

- See also open function

is..... test functions 271–272

- See also — _ctype array

isatty function 273

- See also open function

J

JOIN command

- creating linkable libraries 31–32
- disadvantages 32
- syntax 31

jrand48 function 274

- See also random-number generation functions

L

labs function 275

- See also mathematical macros and functions

lcong48 function 276

- See also random-number generation functions

ldexp function 277

- See also mathematical macros and functions

ldiv function 278–279

- See also div function

libcall #pragma statement 26–27

libcode compiler option 37

LibInit function 38

libraries

- See linkable libraries
- See linkable libraries, custom
- See shared libraries
- See shared libraries, custom

- library bases 7–10
 - #define statement required before referencing 7
 - defining 7–10
 - .device library files 9
 - global variable as library base 7
 - initializing 7–8
 - .library files 8–9
 - .resource library files 10
- .library files
 - initializing library base 7
 - library bases 8–9
- library functions 83–563
 - abort 103
 - abs 104
 - access 105–106
 - acos 107
 - argopt 108–110
 - asctime 111–112
 - asin 113
 - assert 114–115
 - astcsma 116–117
 - atan 118
 - atan2 119
 - atexit 120
 - atof 121–122
 - atoi 123
 - atol 124
 - bldmem 125
 - block I/O functions 96
 - bsearch 126
 - built-in functions 20, 102
 - calling in custom linkable
 - libraries 32–33
 - calling in custom shared
 - libraries 36–37
 - calloc 127–128
 - ceil 129
 - character I/O functions 96
 - character type macros and
 - functions 85–86
 - chdir 130
 - chgetk 131
 - Chk_Abort, chkabort 132
 - chkml 133
 - chkufb 134
 - chmod 135–136
 - clearerr 137
 - clock 138–139
 - close 140
 - closedir 141
 - clrerr 142
 - conversion functions 88–89
 - cos 143
 - cosh 144
 - cot 145
 - creat 146
 - ctime 147
 - _CXFERR 149
 - _CXOVF 150
 - datecmp 151
 - _dclose 152
 - _dcreat 153
 - _dcreatx 154
 - diagnostic control functions 94
 - difftime 157–158
 - directory functions 100–101
 - div 159–160
 - dnex 161–162
 - _dopen 163
 - dqsort 164
 - drand48 165
 - _dread 166
 - _dseek 167
 - _dwrite 168
 - ecvt 169–170
 - _emit 171
 - erand48 172
 - error handling functions 97
 - except 173–174
 - exit 175–176
 - _exit 177
 - exp 178
 - fabs 179
 - fclose 180
 - fcloseall 181
 - fcvt 182–183
 - fdopen 184

- feof 185
- ferror 186
- fflush 187
- fgetc 188
- fgetchar 189
- fgetpos 190–192
- fgets 193–194
- file control functions 97–98
- file management functions 99
- file positioning functions 98
- fileno 195
- findpath 196–197
- floor 198
- flushall 199
- fmod 200
- fmode 201
- fopen 202–206
- forkl, forkv 207–213
- formatted I/O functions 97
- fprintf 214–220
- fputc 221
- fputchar 222
- fputs 223
- fqsort 224
- fread 225
- free 226–228
- freopen 229
- frexp 230
- fscanf 231–233
- fseek 234
- fsetpos 235
- fstat 236–237
- ftell 238
- function categories 84
- fwrite 239
- gcvt 240
- general utility macros and functions 91
- getasn 242–243
- geta4 241
- getc 244
- getcd 245
- getch 246
- getchar 247
- getclk 248
- getcwd 249
- getdfs 250–251
- getenv 252–253
- getfa 254
- getfnl 255–257
- getft 258
- getmem 259
- getml 260
- getpath 261
- getreg 262–263
- gets 264–265
- gmtime 266–267
- halloc 268
- iabs 269
- iomode 270
- is..... test functions 271–272
- isatty 273
- jrand48 274
- labs 275
- lcong48 276
- ldexp 277
- ldiv 278–279
- localeconv 280–281
- localization functions 101
- localtime 282
- log 283
- log10 284
- longjmp 285
- lqsort 286
- lrnd48 287
- lsbrk 288
- lseek 289–290
- lstat 291–292
- main 294–297
- _main 293
- malloc 298–299
- mathematical macros and functions 89–90
- _matherr 300–301
- max 302
- mblen 303
- mbstowcs 304
- mbtowc 305

library functions (*continued*)

- memccpy 306
- memchr 307
- __MemCleanup 308
- memcmp 309
- memcpy 310
- memmove 311–312
- memory allocation functions 93
- memory manipulation
 - functions 94
- memset 313
- min 314
- mkdir 315
- mktime 316–317
- modf 318–319
- movmem 320
- mrnd48 321
- multibyte character functions
 - 101
- nrnd48 322
- offsetof 323–325
- onbreak 326–327
- onexit 328–329
- open 330–331
- opendir 332–333
- ovlyMgr 334
- perror 335
- poserr 336
- pow 339
- pow2 340
- printf 337–338
- program control functions
 - 92–93
- putc 341
- putchar 342
- putenv 343–344
- putreg 345–346
- puts 347
- qsort 348
- raise 349
- rand 350–351
- random-number generation
 - functions 92
- rawcon 352–353
- rbrk 354
- read 355
- readdir 356
- readlocale 357
- realloc 358
- remove 359–360
- rename 361–362
- repmem 363–364
- rewind 365
- rewinddir 366
- rlsmem 367
- rlsml 368
- rmdir 369
- rstmem 370
- sbrk 371
- scanf 372
- seed48 373
- seekdir 374
- setbuf 375
- setjmp 376–377
- setlocale 378–379
- setmem 380
- setnbf 381
- setvbuf 382–383
- signal 384–385
- sin 386
- sinh 387
- sizmem 388
- sorting functions 91
- sprintf 389–390
- sqrt 392
- sqsort 391
- srand 393–394
- srand48 395
- sscanf 396
- stacksize 397
- stackused 398
- standard I/O functions 95
- stat 399–400
- stcarg 401–402
- stccpy 403
- stcd_i 404–405
- stcd_l 406–407
- stcgfe 408–409

- stcgfn 410
- stcgfp 411–412
- stch_i 413–414
- stch_l 415–416
- stci_d 417–418
- stci_h 419–420
- stci_o 421–422
- stcis 423–424
- stciscn 425–426
- stcl_d 427–428
- stcl_h 429–430
- stcl_o 431–432
- stclen 433
- stco_i 434–435
- stco_l 436–437
- stcpm 438–440
- stcpma 441–443
- stcsma 444
- stcu_d 445–446
- stcul_d 447–448
- stpblk 449
- stpbrk 450
- stpchr 451
- stpchrn 452
- stpcpy 453
- stpdata 454–455
- stpsym 456–457
- stptime 458–459
- stptok 460–461
- strbpl 462–463
- strcat 464
- strchr 465
- strcmp 466–468
- strcmpi 469
- strcoll 470
- strcpy 471
- strcspn 472
- strdup 473
- strerror 474
- strftime 475–477
- stricmp 478–479
- _STRICT_ANSI symbol 84
- string functions 86–87
- string I/O functions 96
- strins 480
- strlen 481
- strlwr 482
- strmfe 483
- strmfn 484–485
- strmfp 486
- strmid 487
- strncat 488–489
- strncmp 490–491
- strncpy 492
- strnicmp 493–494
- strnset 495
- strpbrk 496
- strrchr 497–498
- strrev 499
- strset 500
- strsfm 501–502
- strspn 503–504
- strsrt 505–506
- strstr 507–508
- strtod 509–510
- strtok 511–512
- strtol 513–514
- strtoul 515–516
- strupr 517
- strxfrm 518–519
- stspfp 520
- _stub 521
- swmem 522
- system 523
- system interface functions 100
- tan 524
- tanh 525
- tell 526
- telldir 527
- time 528
- time functions 94–95
- _timecvrt 529
- timer 530
- _tinymain 531
- tmpfile 532–533
- tmpnam 534
- toascii, tolower, toupper 535–536

library functions (*continued*)

- qsort 537
- __tzset 538–539
- ungetc 540–541
- unlink 542–543
- user-replaceable functions 101
- utpack 544–545
- utunpk 546–547
- va_arg 548–550
- va_end 551
- va_start 552
- varying-length argument list
 - macros 91
- vfprintf 553–554
- vprintf 555–556
- vsprintf 557–558
- wait, waitm 559
- wctombs 560
- wctomb 561
- write 562
- XCEXIT 563
- linkable libraries 11–17
 - See also linkable libraries, custom
 - base=abs compiler option 12, 15
 - compiling with correct library 12–13
 - definition 1, 31
 - determining which library to use 11–12
 - far keyword 15
 - format for library names 11–12
 - linking with correct library 13–15
 - math=ffp compiler option 13
 - math=ieee compiler option 13
 - math=standard compiler option 13
 - math=68881 compiler option 13
 - near and far data sections 15
 - options required to compile and link with each library

- 14–15
- parms=register compiler option 12
- sc.lib 2
- scm.lib 2, 15
- scmffp.lib 3, 16
- scmieee.lib 3, 16
- scms.lib 3, 16
- scm881.lib 3, 16–17
- scnb.lib 2
- scs.lib 2
- scsnb.lib 2
- shortint compiler option 12, 15
- some compiler options not supported 12
- standard libraries 2
- using linkable libraries 2–3
- using math libraries 15–17
- linkable libraries, custom 31–33
 - calling functions 32–33
 - creating with JOIN command 31–32
 - creating with Object Module Librarian (OML) 31, 32
 - creating with slink or fd2pragma utility 31
 - naming with .lib extension 31, 32
 - when to create 31
- LinkerDB symbol 37–38
- linking 13–15
 - See also slink command
 - compiling and linking with one command 13–14
 - linking custom libraries 32–33
 - options required to compile and link with each library 14–15
 - rules for linking with more than one library 13
- localeconv function 280–281
 - See also setlocale function
- localization functions
 - list of functions 101

localeconv 280–281
 readlocale 357
 setlocale 378–379
 strcoll 470
 strxfrm 518–519
 localtime function 282
 See also time functions
 log function 283
 See also log10 function
 log10 function 284
 See also log function
 longjmp function 285
 See also program control
 functions
 lqsort function 286
 See also sorting functions
 lrand48 function 287
 See also random-number
 generation functions
 lsrbrk function 288
 See also memory allocation
 functions
 lseek function 289–290
 See also open function
 See also tell function
 lstat function 291–292
 See also chmod function

M

magic value for #pragma
 statements 27–29
 main function 294–297
 See also __main function
 See also program control
 functions
 __main function 293
 See also main function
 See also __tinymain function
 makegst compiler option 21
 malloc function 298–299
 See also memory allocation
 functions
 math libraries 15–17
 scm.lib 2, 15

scmffp.lib 3, 16
 scmieee.lib 3, 16
 scms.lib 3, 16
 scm881.lib 3, 16–17
 MathBase 59
 mathematical macros and
 functions
 abs 104
 acos 107
 asin 113
 atan 118
 atan2 119
 ceil 129
 cos 143
 cosh 144
 cot 145
 div 159–160
 exp 178
 fabs 179
 floor 198
 fmod 200
 frexp 230
 iabs 269
 labs 275
 ldexp 277
 ldiv 278–279
 list of functions 89–90
 log 283
 log10 284
 max 302
 min 314
 modf 318–319
 pow 339
 pow2 340
 sin 386
 sinh 387
 sqrt 392
 tan 524
 tanh 525
 math=ffp compiler option 13
 math=ieee compiler option 13
 math=standard compiler option
 13
 math=68881 compiler option 13

—_matherr function 300–301

See also except function

See also _FPERR

MathTranBase 60

max function 302

See also min function

mblen function 303

See also multibyte character functions

mbstowcs function 304

See also multibyte character functions

mbtowc function 305

See also multibyte character functions

memcpy function 306

See also memcpy function

See also strcpy function

memchr function 307

—MemCleanup function 308

memcmp function 309

memcpy function 310

See also memory manipulation functions

See also strcpy function

memmove function 311–312

See also memory manipulation functions

See also strcpy function

memory allocation functions

bldmem 125

calloc 127–128

chkml 133

free 226–228

getmem 259

getml 260

halloc 268

list of functions 93

lsbrk 288

malloc 298–299

—MemCleanup 308

rbrk 354

realloc 358

rlsmem 367

rlsml 368

rstmem 370

sbrk 371

sizmem 388

memory data name 68

memory manipulation functions

list of functions 94

memcpy 306

memchr 307

memcmp 309

memcpy 310

memmove 311–312

memset 313

movmem 320

repmem 363–364

setmem 380

swmem 522

memset function 313

See also setmem function

—MemType 61–62

min function 314

See also max function

mkdir function 315

See also rmdir function

mktime function 316–317

See also time function

modes, file 202–203

modf function 318–319

See also fmod function

—_montbl array 63–64

See also —_daytbl array

See also —_months array

—_months array 65–66

See also —_daytbl array

See also —_montbl array

movmem function 320

See also memmove function

—mprintf function 20

mrnd48 function 321

See also random-number generation functions

—_msflag 67

See also directory functions

`_MSTEP` 68
 See also memory allocation functions
 multibyte character functions
 list of functions 101
`mblen` 303
`mbstowcs` 304
`mbtowc` 305
`wcstombs` 560
`wctomb` 561

N

naming conventions for SAS/C
 libraries 11–12
 near and far data sections 15
`nrnd48` function 322
 See also random-number generation functions
`_nufbs` 69
 See also `_ufbs` array

O

object module 31
 Object Module Librarian (OML)
 31, 32
`offsetof` function 323–325
 OML
 See Object Module Librarian (OML)
`onbreak` function 326–327
 See also program control functions
`onexit` function 328–329
 See also `exit` function
`open` function 330–331
 See also `chmod` function
 See also file control functions
`OpenDevice` function 8
`opendir` function 332–333
 See also directory functions
`OpenLibrary` function 7, 36
`OpenResource` function 7
`_OSERR` 70–73

`ovlyMgr` function 334

P

parent process
 See `forkl`, `forkv` functions
`parms=register` compiler option
 12
 pattern matching 438, 441, 444
 See also string functions
 percent (%) sign, conversion
 specifiers
`strftime` function 475–476
`fprintf` function 214
`fscanf` function 231
`perror` function 335
 See also diagnostic control functions
 See also `errno`
 See also `_sys_errlist`
 See also `_sys_nerr`
`poserr` function 336
 See also diagnostic control functions
 See also `_OSERR`
 pound (#) sign
`fprintf` function 215
 starting instructions in
`fd2pragma` utility 29
`pow` function 339
 See also `pow2` function
`pow2` function 340
 See also `pow` function
`#pragma` statements 25–30
 calculating magic value 27–29
 compared with stub routines 25
 creating 26–27
 generating with `fd2pragma`
 utility 29–30
`libcall` statement 26–27
 register conventions 27–28
`syscall` statement 26, 27
`tagcall` statement 26, 27
 using instead of stub routines 6

predefined data names 41–81
 __BackgroundIO 42
 __Backstdout 43
 __buffsize 44
 __ctype 45–46
 __daytbl 47–48
 DOSBase 49
 errno 50–52
 __fmask 53
 __fmode 54
 __FPERR 55–56
 GfxBase 57
 IntuitionBase 58
 MathBase 59
 MathTranBase 60
 __MemType 61–62
 __monthbl 63–64
 __months 65–66
 __msflag 67
 _MSTEP 68
 __nufbs 69
 __OSERR 70–73
 __priority 74
 __procname 75
 __sigfunc 76
 __stack 77
 __sys_errlist 78
 __sys_nerr 79
 __TZ 80
 __ufbs 81
 printf function 337–338
 See also fprintf function
 using the built-in printf
 function 20
 __priority global variable 74
 See also __procname global
 variable
 See also __stack global
 variable
 process ID structure 209
 processes
 aborting 103
 creating 207
 __procname global variable 75

See also __priority global
 variable
 See also __stack global
 variable
 program control functions
 abort function 103
 atexit 120
 Chk_Abort, chkabort 132
 exit 175–176
 __exit 177
 forkl, forkv 207–213
 list of functions 92–93
 longjmp 285
 onbreak 326–327
 onexit 328–329
 raise 349
 setjmp 376–377
 signal 384–385
 wait, waitm 559
 XCEXIT 563
 prototype files 6–7
 contents of 2
 #pragma statements in 6
 proto/all.h for including all
 header files 7
 putc function 341
 See also fopen function
 putchar function 342
 See also fopen function
 putenv function 343–344
 See also getenv function
 putreg function 345–346
 See also getreg function
 puts function 347
 See also ferror function
 See also fopen function
 See also fputc function
 See also fputs function

Q

qsort function 348
 See also sorting functions

R

raise function 349
 See also program control functions

rand function 350–351
 See also random-number generation functions

random-number generation functions

- drand48 165
- erand48 172
- jrand48 274
- lcong48 276
- list of functions 92
- lrand48 287
- mrnd48 321
- nrnd48 322
- rand 350–351
- seed48 373
- srand 393–394
- srand48 395

rawcon function 352–353
 See also character I/O functions

rbrk function 354
 See also memory allocation functions

read function 355
 See also file control functions
 See also write function

Read mode 202

readdir function 356
 See also directory functions

readlocale function 357
 See also setlocale function

realloc function 358
 See also memory allocation functions

register conventions for #pragma statements 27–28

register keyword 33–34

registers

- assigning values 345
- reading contents 262

remove function 359–360
 See also unlink function

rename function 361–362

repmem function 363–364

.resource library files

- initializing library base 7
- list of library bases 10

rewind function 365
 See also file positioning functions

See also fopen function

rewinddir function 366
 See also directory functions

rlsmem function 367
 See also memory allocation functions

rlsml function 368
 See also memory allocation functions

rmdir function 369
 See also mkdir function

rstmem function 370
 See also memory allocation functions

S

SAS/C libraries

- See linkable libraries
- __saveds keyword 34

sbrk function 371
 See also memory allocation functions

sc command

- See compiling

sc.lib 2

scanf function 372
 See also fscanf function
 See also sscanf function

scm.lib 2, 15

scmffp.lib 3, 16

scmieee.lib 3, 16

scms.lib 3, 16

scm881.lib 3, 16–17

scnb.lib 2

- scs.lib 2
- scsnb.lib 2
- searching an array 126
- seed48 function 373
 - See also random-number generation functions
- seekdir function 374
 - See also directory functions
- setbuf function 375
 - See also fopen function
 - See also file management functions
- setjmp function 376–377
 - See also longjmp function
- setlocale function 378–379
 - See also localization functions
- setmem function 380
 - See also memory manipulation functions
- setnbf function 381
 - See also file management functions
 - See also fopen function
- setvbuf function 382–383
 - See also file management functions
 - See also fopen function
- shared libraries 5–10
 - See also #pragma statements
 - See also shared libraries, custom
 - access methods 2, 5
 - amiga.lib 5–6
 - defining library bases 7–10
 - definition 1
 - including header files 7
 - prototype files 2, 6–7
 - stub routines 5–6
 - using shared libraries 1–2
- shared libraries, custom 33–39
 - calling functions in 36–37
 - creating 33–36
 - format 39
 - global data section 37
 - LibInit function 38
 - library that copies near data section 33
 - library with one near data section 33
 - LinkerDB symbol 37–38
 - modules compiled with libcode compiler option 37–38
 - restrictions 37
 - simultaneous access by multiple processes 37–38
 - slink command and 38
- shortint compiler option
 - definition 12
 - when to use 15
- _sigfunc array 76
 - See also program control functions
- signal function 384–385
- signals
 - establishing event traps 384
 - generating 349
- sin function 386
 - See also mathematical macros and functions
- sinh function 387
 - See also mathematical macros and functions
- sizmem function 388
 - See also memory allocation functions
- _SLASH variable 484, 486
- slashes (/)
 - purpose in function description files 30
- slink command
 - See also linking
 - creating linkable libraries 31–32
 - creating shared libraries 33–36
 - effect on custom shared libraries 38
 - format of libraries produced by 38–39
- sorting functions
 - bsearch 126

- dqsort 164
- fqsort 224
- list of functions 91
- lqsort 286
- qsort 348
- sqsort 391
- strsrt 505–506
- tsort 537
- sprintf function 389–390
 - See also formatted I/O functions
- sqrt function 392
 - See also pow function
 - See also pow2 function
- sqsort function 391
 - See also sorting functions
- srand function 393–394
 - See also random-number generation functions
- srand48 function 395
 - See also random-number generation functions
- sscanf function 396
- _stack global variable 77
 - See also —_priority global variable
 - See also —_procname global variable
- stacksize function 397
 - See also —_stack global variable
- stackused function 398
 - See also —_stack global variable
- standard I/O functions
 - fgetchar 189
 - fputchar 222
 - getch 246
 - getchar 247
 - gets 264–265
 - list of functions 95
 - printf 337–338
 - putchar 342
 - puts 347
 - scanf 372
 - vprintf 555–556
- stat function 399–400
 - See also file management functions
- stat structure 236, 291, 399
- stcarg function 401–402
 - See also string functions
- stccpy function 403
- stcd_i function 404–405
- stcd_l function 406–407
- stcgfe function 408–409
 - See also file management functions
- stcgfn function 410
 - See also file management functions
- stcgfp function 411–412
 - See also file management functions
- stch_i function 413–414
- stch_l function 415–416
- stci_d function 417–418
 - See also conversion functions
- stci_h function 419–420
 - See also conversion functions
- stci_o function 421–422
 - See also conversion functions
- stcis function 423–424
 - See also string functions
- stciscn function 425–426
 - See also string functions
- stcd_d function 427–428
 - See also conversion functions
- stcd_h function 429–430
 - See also conversion functions
- stcd_o function 431–432
 - See also conversion functions
- stclen function 433
 - See also strlen function
- stco_i function 434–435
 - See also conversion functions
- stco_l function 436–437
 - See also conversion functions

- stcpm function 438–440
 - See also string functions
- stcpma function 441–443
 - See also string functions
- stcsma function 444
 - See also string functions
- stcu_d function 445–446
 - See also conversion functions
- stcul_d function 447–448
 - See also conversion functions
- stpbk function 449
 - See also string functions
- stpbrk function 450
 - See also string functions
- stpchr function 451
 - See also string functions
- stpchrn function 452
 - See also string functions
- stpcpy function 453
 - See also string functions
- stupdate function 454–455
 - See also getclk function
 - See also getft function
 - See also stptime function
- stpsym function 456–457
 - See also string functions
- stptime function 458–459
 - See also getclk function
 - See also getft function
 - See also stupdate function
- stptok function 460–461
 - See also string functions
- strbpl function 462–463
 - See also getfnl function
 - See also strsr function
- strcat function 464
 - See also string functions
- strchr function 465
 - See also string functions
- strcmp function 466–468
 - See also string functions
- strcmpi function 469
 - See also string functions
- strcoll function 470
 - See also setlocale function
- strcpy function 471
 - See also string functions
- strcsn function 472
 - See also string functions
- strdup function 473
 - See also malloc function
 - See also strcpy function
- strerror function 474
- strftime function 475–477
 - See also time functions
 - See also setlocale function
- stricmp function 478–479
 - See also string functions
- _STRICT_ANSI symbol 84
- string functions
 - astcsma 116–117
 - list of functions 86–87
 - stcarg 401–402
 - stccpy 403
 - stcis 423–424
 - stciscn 425–426
 - stclen 433
 - stcpm 438–440
 - stcpma 441–443
 - stcsma 444
 - stpbk 449
 - stpbrk 450
 - stpchr 451
 - stpchrn 452
 - stpcpy 453
 - stpsym 456–457
 - stptok 460–461
 - strbpl 462–463
 - strcat 464
 - strchr 465
 - strcmp 466–468
 - strcmpi 469
 - strcpy 471
 - strcsn 472
 - strdup 473
 - stricmp 478–479
 - strins 480
 - strlen 481
 - strmid 487

- strncat 488–489
- strncmp 490–491
- strncpy 492
- strnicmp 493–494
- strnset 495
- strpbrk 496
- strrchr 497–498
- strrev 499
- strset 500
- strspn 503–504
- strstr 507–508
- strtok 511–512
- string I/O functions
 - fgets 193–194
 - fputs 223
 - gets 264–265
 - list of functions 96
 - puts 347
- strins function 480
 - See also strcat function
- strlen function 481
 - See also stclen function
- strlwr function 482
 - See also conversion functions
 - See also stricmp function
- strmfe function 483
 - See also file management functions
- strmfnc function 484–485
 - See also file management functions
- strmfp function 486
 - See also file management functions
- strmid function 487
 - See also string functions
- strncat function 488–489
 - See also string functions
- strncmp function 490–491
 - See also string functions
- strncpy function 492
 - See also string functions
- strnicmp function 493–494
 - See also string functions
- strnset function 495
 - See also strset function
- strpbrk function 496
 - See also string functions
- strrchr function 497–498
 - See also string functions
- strrev function 499
- strset function 500
- strsfnc function 501–502
 - See also file management functions
- strspn function 503–504
 - See also string functions
- strsrt function 505–506
 - See also getfnl function
 - See also strbpl function
 - See also tqsort function
- strstr function 507–508
 - See also string functions
- strtod function 509–510
 - See also scanf function
- strtok function 511–512
 - See also string functions
- strtol function 513–514
 - See also strtoul function
- strtoul function 515–516
 - See also strtol function
- structures
 - FORKENV 208
 - process ID 209
 - TermMsg 208–209
- strupr function 517
 - See also conversion functions
 - See also stricmp function
- strxfrm function 518–519
 - See also localization functions
- stspfp function 520
 - See also file management functions
- _stub function 521
- stub routines
 - accessing library functions 5–6
 - compared with #pragma statements 25

stub routines (*continued*)
 using #pragma statements instead of 6
 swmem function 522
 — __sys_errlist 78
 See also diagnostic control functions
 See also — __sys_nerr
 — __sys_nerr 79
 See also diagnostic control functions
 See also — __sys_errlist
 syscall #pragma statement 26, 27
 system function 523
 system header files 19–23
 built-in functions 20
 compressing 19–20
 creating GSTs 21–22
 definition 19
 global symbol tables (GSTs) 21–23
 including all header files 7
 including #pragma statements 6
 restrictions for use in GSTs 22
 system interface functions
 argopt 108–110
 chgclk 131
 getasn 242–243
 getclk 248
 getdfs 250–251
 getenv 252–253
 list of functions 100
 putenv 343–344
 rawcon 352–353
 stacksize 397
 stackused 398
 system 523

T

tagcall #pragma statement 26, 27
 tan function 524
 tanh function 525
 tell function 526
 See also file positioning

 functions
 See also open function
 telldir function 527
 See also directory functions
 TermMsg structure 208–209
 time function 528
 See also time functions
 time functions
 asctime 111–112
 clock 138–139
 ctime 147–148
 datecmp 151
 difftime 157–158
 gmtime 266–267
 list of functions 94–95
 localtime 282
 mktime 316–317
 strftime 475–477
 time 528
 — _timecv 529
 timer 530
 — _tzset 538–539
 utpack 544–545
 utunpk 546–547
 — _timecv function 529
 See also mktime function
 timer function 530
 — _tinymain function 531
 See also main function
 _tinyprintf function 20
 tmpfile function 532–533
 See also file control functions
 tmpnam function 534
 See also tmpfile function
 toascii, tolower, toupper functions 535–536
 See also — _ctype array
 See also is..... test functions
 tqsort function 537
 See also sorting functions
 Translate mode 203
 traps 120, 384
 Trunc mode 202
 _TZ variable 80
 See also — _tzset function

— `_tzset` function 538–539
 See also `localtime` function

U

UFB structure 81
 — `_ufbs` array 81
 See also `chkufb` function
 See also — `_nufbs`
 See also `open` function
`#undef` statement 20
`ungetc` function 540–541
`unlink` function 542–543
 See also `remove` function
 user-replaceable functions
 `_CXFERR` 149
 `_CXOVF` 150
 — `_UserLibInit` routine 34, 37
`utpack` function 544–545
 See also `getcl` function
 See also `stupdate` function
 See also time functions
`utunpk` function 546–547
 See also `getcl` function
 See also `stupdate` function
 See also time functions

V

`va_arg` macro 548–550
 See also varying-length argument
 list macros
`va_end` macro 551
 See also varying-length argument
 list macros
`va_start` macro 552
 See also varying-length argument
 list macros
 varying-length argument list
 macros
 list of macros 91
 `va_arg` 548–550
 `va_end` 551
 `va_start` 552
`vfprintf` function 553–554
 See also `fprintf` function

`vprintf` function 555–556
 See also `printf` function
`vsprintf` function 557–558
 See also `sprintf` function

W

`wait`, `waitm` functions 559
 See also program control
 functions
`wcstombs` function 560
 See also multibyte character
 functions
`wctomb` function 561
 See also multibyte character
 functions
 wide characters 560, 561
 wildcard characters for specifying
 patterns 438, 441
 with files 6
`write` function 562
 See also block I/O functions
 See also file control functions
 Write mode 203

X

`XCEXIT` function 563
 See also — `_exit` function

Special Characters

`*` (asterisk)
 See asterisk (`*`)
`[]` (brackets)
 See brackets (`[]`), `fprintf`
 function
`,` (comma)
 See comma (`,`)
`%` (percent) sign
 See percent (`%`) sign, conversion
 specifiers
`#` (pound) sign
 See pound (`#`) sign
`/` (slashes)
 See slashes (`/`)

Your Turn

If you have comments or suggestions about *SAS/C Development System Library Reference, Version 6.0* or the SAS/C Development System, please send them to us on a photocopy of this page.

Please return the photocopy to the Publications Division (for comments about this book) or the Technical Support Division (for suggestions about the SAS/C Development System) at SAS Institute Inc., SAS Campus Drive, Cary, NC 27513.

include <graphics/gfxbase.h>
Openlibrary("intuition.library"
Library("graphics.library"

Late changes required reprinting some pages of SAS/C Development System Library Reference, Version 6.0. As you assemble the SAS/C Development System Library Reference binder, please use the pages included here to replace the like-numbered pages in the manual.

**Replacement
Pages for
SAS/C® Development System
Library Reference
Version 6.0**

3 Using the SAS/C® Libraries

- 11 *Introduction*
- 11 *Determining Which Library to Use*
- 12 *Compiling with the Correct Library*
- 13 *Linking with the Correct Libraries*
- 15 *Using the Math Libraries*

Introduction

As described in Chapter 1, “What Libraries Can I Access?,” the SAS/C Development System includes several libraries. The names of these libraries indicate the type of routines in the library and, therefore, the compiler options you must use with the libraries. The following sections describe the naming conventions for SAS/C libraries and the compiler options to use with each library.

Determining Which Library to Use

If you know what type of library routines you need to use, you can determine the library you need to use by looking at the letters included in the library name. The names of the SAS/C libraries follow this format:

```
sc[s][nb][m[ieee | ffp | 881]].lib
```

where

- s** indicates the library uses short integers.
- nb** indicates the library uses far addressing (no base register).
- m** indicates the library uses floating-point math. If the library is a math library, the library name extension indicates the type of floating-point routines supported by the library, as follows:
 - none **scm.lib** contains standard SAS/C floating-point routines.

- ieee** The library contains Commodore IEEE routines.
- ffp** The library uses the Motorola fast floating-point format.
- 881** The library uses Motorola 68881 instructions.

Note: Each type of math library is mutually exclusive. You can use only one math library at a time.

For example, **scnb.lib** uses far addressing, **scs.lib** uses 16-bit integers, and **scsnb.lib** uses far addressing and 16-bit integers.

Because of disk space considerations, all possible libraries are not included in this software. The libraries supplied in this software are described under “Using the Linkable Libraries” in Chapter 1.

Because not all possible libraries are included, some combinations of compiler options are not supported. For example, combining the use of short integers with 68881 instructions would require a library named **scms881.lib**, which is not supplied with this software. Therefore, the combination of compiler options **shortint** and **math=68881** is not supported. In addition, the combination of options **shortint** and **math=ieee** (**scmsieee.lib**) is not supported.

If you need a library that is not provided with the SAS/C Development System, contact the Technical Support Division at SAS Institute.

Compiling with the Correct Library

Certain compiler options tell the compiler which SAS/C library to use. The compiler chooses the correct libraries based on the option or combination of options you specify.

The following lists show the compiler options you should specify for the types of library routines you need to use.

parms=register

tells the compiler to use registerized parameters.*

shortint

tells the compiler to use short integers.

base=abs

tells the compiler to use absolute data addressing (no base register).

* The standard libraries now provide support for registerized parameters. To use registerized parameters, you must compile with the **parms=register** compiler option, but you do not need to link with a special library.

The **math=** options are mutually exclusive and indicate the type of floating-point math to be used, as follows:

math=standard

tells the compiler to use standard SAS/C floating-point routines.

math=ieee

tells the compiler to use Commodore IEEE routines.

math=ffp

tells the compiler to use the Motorola fast floating-point format.

math=68881

tells the compiler to use Motorola 68881 instructions.

Linking with the Correct Libraries

You can run the compiler and linker (**slink**) as separate steps (call each tool explicitly), or you can specify the **link** option in the **sc** command to link your program automatically after it compiles.

If you call **slink** separately, you must specify each library explicitly after the **lib** keyword. Make sure you link your libraries in the correct order. When you are linking more than one library, follow these rules:

1. Specify any libraries you create first.
2. Specify your math library.
3. Specify any additional nonmath libraries.
4. Specify **amiga.lib** last.

For example, to compile and link **prog.c** using short integers, you could call the compiler and linker separately as follows:

```
sc shortint prog.c
slink lib:c.o+prog.o to test lib lib:scs.lib lib:amiga.lib
```

If you want to compile and link your program with one command, enter the **link** option in addition to any other options you need on the **sc** command line. If the compiler does not find any errors in your program, it will look at the other options you specify, determine which libraries are needed, and call **slink** to link your program. The compiler will link the libraries in the correct order. The compiler uses the standard C library **sc.lib** if you specify **link** without any additional library options. If you specify **math=standard** and **link** without any additional library options, the compiler uses **sc.lib** and the math library **scm.lib**.

For example, to compile and link `prog.c` with one command, use the `link` compiler option as follows:

```
sc shortint link prog.c
```

To compile and link a program using the standard C library, you can use the following command:

```
sc link prog.c
```

If `prog.c` uses floating-point math and you want to use only the standard floating-point library (`scm.lib`), specify the `math=standard` option also:

```
sc math=standard link prog.c
```

To summarize, Table 3.1 shows the options you should use on the `sc` command line to compile and link with each library supplied with the SAS/C Development System.

Table 3.1
*Options Required to
Compile and Link with
Each Library*

Library	Options	Type of Library
<code>sc.lib</code>	<code>link</code>	Standard C library
<code>scs.lib</code>	<code>shortint</code> <code>link</code>	Version of <code>sc.lib</code> that uses 16-bit integers
<code>scnb.lib</code>	<code>base=abs</code> <code>link</code>	Version of <code>sc.lib</code> that uses far addressing
<code>scsnb.lib</code>	<code>shortint</code> <code>base=abs</code> <code>link</code>	Version of <code>sc.lib</code> that uses 16-bit integers and far addressing
<code>scm.lib</code>	<code>math=standard</code> <code>link</code>	Standard math library
<code>scmieee.lib</code>	<code>math=ieee</code> <code>link</code>	IEEE math library supplied by Commodore

(continued)

Table 3.1
(continued)

Library	Options	Type of Library
<code>scmffp.lib</code>	<code>math=ffp</code> <code>link</code>	Motorola fast floating-point math library
<code>scms.lib</code>	<code>math=standard</code> <code>shortint</code> <code>link</code>	Math library for use with 16- bit integers
<code>scm881.lib</code>	<code>math=68881</code> <code>link</code>	68881 coprocessor math library

If you are porting code from a machine with short (16-bit) integers, you should compile with the `shortint` option. The `shortint` option tells the compiler to use 16-bit integers instead of the normal 32 bits. Any code compiled with the `shortint` option must be linked with a library whose name contains the letter `s` after the `sc`.

It is recommended that you use the `far` keyword in the declarations and definitions of some of your larger data structures to move them into the far data section. Continue adding the `far` keyword until the near data section is less than 64K. Alternatively, if your program has no large data structure, you may want to use `base=abs`, signifying long (32-bit) references, to force all data into the far section. For more information on near and far data sections, refer to Chapter 12, “How Does the Compiler Work?” in the *SAS/C Development System User’s Guide, Volume I: Introduction, Compiler, Editor*.

Using the Math Libraries

The standard math library, `scm.lib`, contains all of the floating-point routines described in Chapter 7, “Library Reference.” The other math libraries contain special versions of the routines in `scm.lib`.

The following list contains additional information about using the math libraries.

`scm.lib`

is the standard math library. You must specify a math library if your program performs floating point operations. To use this library, specify the `math=standard` option on the compiler command line. This library ignores any math coprocessors that may be present.

scmieee.lib

calls the IEEE shared math libraries supplied by Commodore to compute floating-point operations. Advantages of using this library include:

- smaller executable code size. Code size is reduced because much of the code needed to perform the floating-point operations is in the shared library.
- optional 68881 or 68882 support. The Commodore IEEE library uses the 68881 or 68882 chip, if it is present, thereby making your program run much faster. If a math coprocessor chip is not present, your program will still run.

The Commodore routines are slightly slower than their counterparts in **scm.lib** if the 68881 coprocessor is not present.

scmffp.lib

performs its operations in Motorola Fast Floating-Point (FFP) format. The FFP routines are faster than the IEEE routines, but they obtain this speed at the expense of accuracy. The precision of an FFP double is half that of an IEEE double. However, for many applications, FFP accuracy is sufficient.

The compiler manipulates real numbers in the FFP format if you specify the **math=ffp** option. Do not mix the FFP and IEEE formats. If one module in an executable uses the **math=ffp** option, then all modules must use it.

This library calls **mathffp.library** and **mathtrans.library**, which are supplied by Commodore.

scms.lib

is the same as the standard math library except it uses 16-bit integers.

scm881.lib

is the SAS/C 68881 Coprocessor Math Library. To get the best performance from this library, include the **m68881.h** header file (**#include <m68881.h>**) and compile with the **math=68881** option. Programs linked with this library will not run on a machine without a 68881 coprocessor.

If you call floating-point math routines in your program, but you have not compiled or linked with the correct options, you will get messages such as:

```
Undefined Symbol: _CXxnn
```

To correct the problem, specify the correct compiler option for the library you need: **math=standard**, **math=ieee**, **math=ffp**, or

Suppose the function `myfunc` has two parameters followed by any optional taglist parameters. You can use `myfunc` with a `#pragma tagcall` statement with a magic value of 98103. The first parameter would be placed into register D1, the second into register A0, all remaining parameters would be placed on the stack, and the address of the first parameter on the stack would be placed into register A1.

Generating #pragma Statements with the fd2pragma Utility

The format of the `#pragma` statements is relatively easy to understand and code but difficult to maintain and or decode. However, you can create a standard ASCII function description (`.fd`) file, and use the `fd2pragma` utility to create a header file containing the corresponding `#pragma` statements.

Creating a function description (.fd) file

A function description file consists of instructions (*directives*) to the `fd2pragma` utility, the entry points of each library function, and comments. You can enter only one statement per line. Comments begin with an asterisk (*); instructions to the `fd2pragma` utility begin with two pound signs (##); any other line is an entry point into a library function.

For example, the following lines are from the `dos_lib.fd` file:

```
* "dos.library"
##base DOSBase
##bias 30
##public
Open(name,accessMode)(d1/d2)
Close(file)(d1)
Read(file,buffer,length)(d1/d2/d3)
*
##end
```

The `fd2pragma` utility accepts the following instructions:

##base name	identifies the global variable pointing to the base of the library.
##bias n	identifies the offset value between the library base and the entry point of the first function in the library. Enter 30 for this value.
##public	identifies subsequent functions as being available to your programs.
##private	identifies subsequent functions as being available to the library only. Programs cannot call functions designated as private. ##private is required for

externally unavailable functions because the offsets are calculated from the positions of the function names in the lists in the `.fd` file. Skipping a private function in the list would cause incorrect offsets to be calculated for subsequent functions.

##end Marks the end of the file.

You must include entry points for all of the functions in your library. The entry points follow this format:

entryname(arguments)(registers)

Use commas to separate the arguments. Use commas or slashes (/) to separate registers.

Note: Commas are used to separate entries that need to be pushed with separate `move` or `movem` assembler instructions. Slashes indicate those entries that can be combined in a single `movem` instruction.

For example, the following lines are from the `intuition.library` file:

```
InitCode(startClass,version)(D0/D1)
PrintIText(rp,itext,left,top)(A0/A1,D0/D1)
```

Running the `fd2pragma` utility

To run the `fd2pragma` utility, enter the `fd2pragma` command as follows:

```
fd2pragma infile.fd newheader.h
```

where

infile specifies the function description file that you created. The `.fd` extension is required.

newheader specifies the file where you want the new `#pragma` statements stored. The `.h` extension is required.

If the `fd2pragma` utility encounters any errors, the errors are printed in the standard output file. For example, your function description file could contain invalid directives, invalid register specifications, or no `##end` statement.

Using the `fd2pragma` utility is less likely to lead to errors than trying to amend the `#pragma` statements themselves. It is strongly suggested that you always maintain the function description file and use the `fd2pragma` utility.

6 Predefined Data Name Reference

This chapter describes the predefined data names provided with the SAS/C software. These data names are external variables that you can read or write from your program. You can also form a pointer to any external name using the ampersand (&) operator.

The data names are described in alphabetical order. Each data name description includes the following sections (if applicable):

- Synopsis* shows the declaration for the data name, including any header files that must be included in the calling routine.
- Description* describes what the data name does, what it is used for, and which functions use it. This section also contains any additional details you need to use the data name.
- Portability* describes the systems or languages with which the data name is compatible. A data name's portability falls into one of these categories:
 - AmigaDOS indicates the data name can be used only under Commodore's AmigaDOS.
 - ANSI indicates the data name is compatible with the definition in the ANSI Standard for C.
 - SAS/C indicates the name is in the SAS/C portable library. These data names are available only when using the SAS/C libraries and may or may not be equivalent to data names in libraries from other vendors.
 - UNIX indicates the data name is compatible with AT&T's UNIX System V or BSD 4.x UNIX.
- Example* contains sample code showing how to use the data name.
- See Also* lists related functions and data names.

__BackgroundIO Route I/O to the console window

Synopsis `long __BackgroundIO;`

Description The global variable `__BackgroundIO` is used by `cback.o` to indicate whether output will be sent to the console window by your background process. The default is 0, indicating that I/O will not be sent to the console window.

Setting `__BackgroundIO` to 1 tells `cback.o` to initialize `_Backstdout` to point to the console window. If `__BackgroundIO` is set to 1, you must call `Close` on `_Backstdout` before the program exits:

```
Close (_Backstdout);
```

If you do not include this statement before your program exits, the CLI window pointed to by `_Backstdout` will not close properly.

This variable is of interest only for background processes. If you do not link with `cback.o`, this variable is ignored.

Portability SAS/C

See Also `_Backstdout`

`__Backstdout` Pointer to the console window file handle

Synopsis `BPTR __Backstdout;`

Description The global variable `__BackgroundIO` is used by `cback.o` to point to the console window where I/O is to be routed. If `__BackgroundIO` is set to 1, `cback.o` initializes `__Backstdout` to point to the console window. Otherwise, `__Backstdout` is not initialized. If `__BackgroundIO` is set to 1 and `__Backstdout` is initialized, you must call `Close` for `__Backstdout` before the program exits.

```
Close (__Backstdout);
```

If you do not include this statement before your program exits, the CLI window pointed to by `__Backstdout` will not close properly.

This variable is of interest only for background processes. If you do not link with `cback.o`, this variable is ignored.

Portability SAS/C

See Also `__BackgroundIO`

— **__buffsize** Level 2 I/O buffer size

Synopsis `extern int __buffsize;`

Description This external integer is used by the level 2 I/O system to specify the size of the buffers allocated for level 2 files. This location is also used to specify the size of a buffer attached to a file with the **setbuf** or **setvbuf** function. **__buffsize** must be set to the size of the buffer before the **setbuf** or **setvbuf** function is called or before the first I/O operation; otherwise, the default buffer size is used.

The buffer is not allocated when the file is opened. Instead, the first I/O operation causes the buffer to be allocated from the local memory pool if one has not been previously specified with the **setbuf** or **setvbuf** function. This means that if **__buffsize** is changed between the **fopen** call and the first I/O operation, the size of the buffer allocated for the file will be the value of **__buffsize** at the time of the I/O operation, not the value when the file was opened.

In Version 5.10 and earlier, this variable was named **_bufsiz**.

Portability SAS/C

See Also **fopen**, **setbuf**

`__FPERR` Floating-point error code

Synopsis `#include <math.h>`

Description `__FPERR` contains a nonzero value after any low-level floating-point operation encounters an error. Low-level operations include addition, subtraction, multiplication, division, comparison, and conversion from one number representation to another (for example, floating point to double-precision floating point).

The `math.h` file contains the declaration `extern int __FPERR`, so you do not need to include this statement in your program.

The following table lists the error codes and their corresponding symbols from the `math.h` file:

Symbol	Value	Meaning
<code>__FPEUND</code>	1	underflow
<code>__FPEOVF</code>	2	overflow
<code>__FPEZDV</code>	3	divide by zero
<code>__FPENAN</code>	4	not a valid number
<code>__FPECOM</code>	5	not comparable

When the error occurs, the low-level operation passes the appropriate error code to the `__CXFERR` function, which must store the code in `__FPERR`. `__FPERR` is never reset by any low-level operation.

For compatibility with previous releases, the error code values without the leading underscore are also defined in the `math.h` file unless you define the `__STRICT_ANSI` flag before including `math.h`.

Note: This variable is not set when using the standard math libraries `scm.lib` and `scms.lib`.

Portability SAS/C

Example

```

/*
 * This example uses the division operation to produce
 * floating point errors. For example, Enter 0 for the
 * divisor and 1 for the dividend to produce
 * __FPERR = 3 (Divide by zero).
 */

```


__FPERR Floating-point error code
(continued)

```
#include <math.h>
#include <stdio.h>

void main (void)
{
    double a,b,c;

    while(feof(stdin) == 0)
    {
        printf("Enter divisor: ");
        if(scanf("%lf",&a) != 1)
            exit(0);

        printf("Enter dividend: ");
        if(scanf("%lf",&b) != 1)
            exit(0);

        __FPERR = 0;

        c = b / a;
        printf("__FPERR = %d\n",__FPERR);
        printf("%lf / %lf = %lf\n\n",b,a,c);
    }
}
```

See Also `errno`

abort Abort the current process

Synopsis `#include <stdlib.h>`

```
void abort(void);
```

Description This function aborts the current process and returns a completion code of 3 to the parent process. Also, the message **Abnormal program termination** is sent to `stderr`. Level 2 I/O buffers are not flushed.

Portability ANSI

Example

```
/*
 * Do your own memory allocation
 */
#include <stdio.h>
#include <stdlib.h>

void *gmem(size_t size)
{
    void *p;

    if ((p=malloc(size))==NULL)
    {
        abort();
    }
    return(p);
}

void main(void)
{
    void *p;

    p = gmem(4096);
    if (p)
    {
        free(p);
    }
}
```

See Also `exit`, `__exit`, `raise`, `_XCEXIT`

abs Absolute value

Synopsis `#include <stdlib.h>`

```
ax = abs(x);
```

```
type x;
```

```
type ax;
```

Description This macro computes the absolute value of the various numeric data types. It accepts any data type as its argument and generates in-line code to perform the conversion. The definition is

```
#define abs(x) ((x)<0?-(x):(x))
```

To minimize code size, you can use the **fabs**, **iabs**, or **labs** function instead. Choose the function that corresponds to the data type being converted.

Portability ANSI

Returns The return value is the absolute value of the argument.

See Also **fabs**, **iabs**, **labs**

clearerr Clear a level 2 I/O error flag

Synopsis `#include <stdio.h>`

`void clearerr(fp);`

`FILE *fp;            /* file pointer */`

Description This macro clears the error flag and end-of-file flag of the file pointed to by the `fp` pointer. Once set, the error flag forces an end-of-file value to be returned any time the file is accessed until the flag is reset.

Portability ANSI

See Also `clrerr`, `ferror`

clock Measure program processor time

Synopsis `#include <time.h>`

```
x = clock(void);
```

```
clock_t x;
```

Description The `clock` function returns the amount of processor time used by the program, relative to the base point. The *base point* for the `clock` function is the value returned by the first call to `clock`. The first call to the `clock` function always returns 0.0.

You can use the `clock` function to determine the amount of processor time used by calling the `clock` function twice in the same program and calculating the difference between the values returned.

The value returned is of type `clock_t`. The value returned is in fractions of a second, where a value of `CLOCKS_PER_SEC` represents one second of processor time. (`clock_t` and `CLOCKS_PER_SEC` are defined in the file `time.h`.) In this implementation, `clock_t` is defined as an unsigned long integer and `CLOCKS_PER_SEC` is 1.

The value returned by the `clock` function is of relatively low accuracy and may depend on the extent of other system activity. Values returned by the `clock` function are likely to be inconsistent from one execution of a program to another.

Portability ANSI

Returns The `clock` function returns the number of seconds since the base time. If an accurate value cannot be returned, `(clock_t)-1` is returned.

Example

```
/*
 * This example determines the processor time
 * required to compute 1000 logarithms.
 */

#include <time.h>
#include <math.h>
#include <stdio.h>

void main(void)
{
    clock_t start, end;
    double index;
```

cot Cotangent function

Synopsis `#include <math.h>`

```
r = cot(x);

double r;    /* result */
double x;    /* angle */
```

Description The `cot` function computes the cotangent of an angle measured in radians.

 This function is not available if the `__STRICT_ANSI` flag has been defined.

Portability UNIX

Returns This function returns the cotangent of the argument expressed in radians.

See Also `__matherr`

creat Create a level 1 file

Synopsis `#include <fnctl.h>`

```
fh = creat(name,prot);

int fh;           /* file handle */
const char *name; /* file name */
int prot;         /* protection mode */
```

Description This function is exactly the same as calling the **open** function in the following way:

```
open(name,O_WRONLY | O_TRUNC | O_CREAT , prot)
```

The file is created if it does not exist and truncated if it does exist. Then, it is opened for writing. The protection mode can be any of the following:

Value	Meaning
S_IWRITE	write permission
S_IREAD	read permission
S_IWRITE S_IREAD	write and read permission
0	write and read permission

In the current AmigaDOS implementation, the protection mode is ignored.

Portability UNIX

Returns If the operation is successful, the function returns a file handle, which is an integer equal to or greater than 0. Otherwise, it returns `-1` and places error information in the external integers **errno** and **_OSERR**.

See Also **chmod**, **close**, **errno**, **open**, **_OSERR**

__CXFERR Low-level floating-point error

Synopsis `#include <math.h>`

```
void __CXFERR(code);
int code;
```

Description This function is called when an error is detected by one of the low-level floating-point routines, such as the arithmetic operations. Higher-level routines, such as the trigonometric functions, use the `__matherr` function.

You cannot call this function. However, you can replace this error trap with an application-dependent routine, as long as it stores the error code in the global integer `_FPERR`. Some of the math functions check the `_FPERR` integer to see if low-level errors occurred. To replace this function, include a prototype as well as the error trap function in your source code. The function and prototype should be named `__CXFERR`.

The error code passed to the `__CXFERR` function indicates the type of floating-point anomaly that occurred, as follows:

Symbol	Value	Meaning
<code>_FPEUND</code>	1	underflow
<code>_FPEOVF</code>	2	overflow
<code>_FPEZDV</code>	3	divide by zero
<code>_FPENAN</code>	4	not a number
<code>_FPECOM</code>	5	not comparable

If the `__STRICT_ANSI` flag has not been defined, equivalent names without the leading underscore are also defined for compatibility with previous releases of the compiler.

These codes are defined in the file `math.h`.

This routine was named `CXFERR` in Version 5 and previous versions.

Portability SAS/C

See Also `__matherr`

__CXOVF Default stack-overflow handler

Synopsis `__CXOVF(void);`

Description This function is the default stack-overflow handler. When called, it puts up a requester indicating the stack-overflow state for the current program. After the user clicks on the requester, this routine terminates the program.

This function is automatically invoked by the stack-overflow detection code. You cannot call this function. However, you can replace this function with your own replacement function. To replace the `__CXOVF` function, include a prototype as well as the function in your source code. The function and prototype should be named `__CXOVF`.

Sample source code is provided in the source directory.

This routine was named `cxovf` in Version 5 and previous versions.

Portability SAS/C

Example

```

/*
 * This is a replacement overflow handler that simply
 * restarts the program at a safe known spot.
 */
#include <jmpbuf.h>

extern jmp_buf safe_spot;
void __stdargs __CXOVF(void)
{
    longjmp(safe_spot, 1);
}
```

except Call the math error handler function

Synopsis `#include <math.h>`

```
r = except(type,name,arg1,arg2,retval);

double r;           /* actual return value */
int type;           /* error type */
const char *name;    /* math function name */
double arg1;         /* first argument */
double arg2;         /* second argument */
double retval;       /* proposed return value */
```

Description The **except** function is a SAS/C extension to UNIX that simplifies the interface to the `__matherr` function by setting up the exception vector and processing the action code and return value. It is intended to ease the error-handling chore in user-written math functions.

When your math function encounters an error, it should call the **except** function specifying one of the following error types, which are defined in the `math.h` header file:

Symbol	Code	Meaning
<code>__DOMAIN</code>	1	domain error
<code>__SING</code>	2	singularity
<code>__OVERFLOW</code>	3	overflow (number too large)
<code>__UNDERFLOW</code>	4	underflow (number too small)
<code>__TLOSS</code>	5	total loss of significance
<code>__PLOSS</code>	6	partial loss of significance

If the `__STRICT_ANSI` flag has been defined, you must use the alternate entry point `__except`. If the `__STRICT_ANSI` flag has not been defined, equivalent names without the leading underscore are also defined for compatibility with previous releases of the compiler.

Portability SAS/C

except Call the math error handler function
(continued)

Returns The actual return value (a double-precision floating-point value) is passed back.

See Also `__fperr`, `__matherr`

fopen Open a level 2 file
(continued)

Mode	Value	Effect
Write	yes	The file can be written with functions such as fwrite and fputc . Also, the fseek function can be used to position the file before writing.
	no	The file cannot be written, but see Append below.
Append	yes	The file can be written, but it is automatically positioned to the current end-of-file before each write operation. This mode prevents existing data from being changed.
	no	Automatic positioning to the end-of-file is not done before a write operation. Also, writes are not allowed unless the Write mode value is Yes .
Translate	default	The external integer _fmode is used to set mode A or mode B as follows: <pre>if (_fmode & 0x8000) set mode B else set mode A</pre> For AmigaDOS, the external integer _fmode is normally 0x8000.
	mode A	On a read operation, each carriage-return character (\r) is deleted, and the Control-Z character is treated as a logical end-of-file mark. On a write operation, each line-feed character (\n) is expanded to a carriage return followed by a line feed.
	mode B	The data are unchanged as they are read or written.

fopen Open a level 2 file
(continued)

The following table shows the list of values specified by each mode string.

Mode String	Create	Trunc	Read	Write	Append	Translate
r	no	no	yes	no	no	default
w	yes	yes	no	yes	no	default
a	yes	no	no	no	yes	default
r+	no	no	yes	yes	no	default
w+	yes	yes	yes	yes	no	default
a+	yes	no	yes	no	yes	default
ra	no	no	yes	no	no	mode A
wa	yes	yes	no	yes	no	mode A
aa	yes	no	no	no	yes	mode A
ra+	no	no	yes	yes	no	mode A
wa+	yes	yes	yes	yes	no	mode A
aa+	yes	no	yes	no	yes	mode A
rb	no	no	yes	no	no	mode B
wb	yes	yes	no	yes	no	mode B
ab	yes	no	no	no	yes	mode B
rb+	no	no	yes	yes	no	mode B
wb+	yes	yes	yes	yes	no	mode B
ab+	yes	no	yes	no	yes	mode B

If the file is successfully opened, the function returns a pointer to a *buffered I/O control block*, which is defined in the header file `stdio.h`. Normally, you will not need to access any information in the control block directly, but you should be very careful not to disturb the block

— **__main** Standard preprocessing for the main module

Synopsis `#include <stdlib.h>`

```
void __stdargs __main(line);
```

```
char *line; /* ptr to command line that caused execution */
```

Description The `__main` function performs the standard preprocessing for the main module of a C program. It accepts a command line of the following form:

```
pgmname arg1 arg2
```

It builds a list of pointers to each argument and the first pointer is to the program name. The `__main` function also opens the standard I/O files `stdin`, `stdout`, and `stderr`. `__main` calls the function `main` with the standard `argc` and `argv` parameters.

Unlike previous editions of the compiler, this function is declared with the `__stdargs` keyword.

Note: This function is declared with two underscores in C code, but the actual symbol has three underscores.

The source code for this function is in the file `_main.c` in the `sc:source` directory.

Portability SAS/C

See Also `main`, `__tinymain`

main Your main or principal function

Synopsis `#include "workbench/startup.h"`

```
void main(argc,argv);

int argc;           /* argument count */
union {
    char *args[];
    struct WBStartup *msg;
} argv;             /* argument vector */
```

Description This function does not actually exist in the library; you must supply one of these main programs in each of your applications. If you trace through the two startup modules `c.a` and `_main.c`, you will find that the module `c.a` passes control to the module `_main.c`, which then calls the function named `main`. Since the source code for both of these modules is supplied, you are free to change this initialization procedure for special applications. The standard version simulates UNIX's interface with C programs by setting up a vector, which is simply an array of pointers.

The `argv.args` array contains pointers to the command-line arguments, and the argument `argc` indicates how many pointers are in the array. For example, you can start the program `myprog` with the following command:

```
myprog abc def "ghi jkl"
```

Then, set up the `argv.args` array as follows:

```
argv.args[0] => "myprog"
argv.args[1] => "abc"
argv.args[2] => "def"
argv.args[3] => "ghi jkl"
```

The `argc` argument contains the value 4.

Under Workbench, there is no command line. In this case, the argument `argc` is 0 indicating no command or arguments, and the argument `argv.msg` is a pointer to the Workbench startup message structure.

Portability ANSI

main Your main or principal function
(continued)

Returns When the **main** function returns to its caller (normally the `_main.c` function), the program exits to AmigaDOS with a termination code of 0. If you want to pass a nonzero termination code back to AmigaDOS, use the **exit** or **__exit** function.

Example

```

/**
 * This program is intended to run only under CLI
 * and displays the command and any arguments
 */

void main(int argc, char *argv[])
{
    int i;

    printf("command = %s\n",argv[0]);
    for (i = 0; argc > 1; i++, argc--)
    {
        printf("argument %d = %s\n",i,argv[i]);
    }
}

/**
 * This program is intended to run only under WorkBench and
 * gets its arguments from the WorkBench message structure
 */

#include <workbench/startup.h>

void main(int argc, char *argv[])
{
    struct WBStartup *wbs;
    int i;

    if (argc != 0)
    {
        exit(EXIT_FAILURE);
    }

    wbs = (struct WBStartup)argv;

```


main Your main or principal function
(continued)

```

    printf("command = %s\n", wbs->sm_ArgList[0].wa_Name);
    for (i = 1; i < wbs->sm_NumArgs; i++)
        printf("argument %d = %s\n",
            i, wbs->sm_ArgList[i].wa_Name);
}

/*
 * This program is intended to run under either
 * WorkBench or CLI.
 */

#include <stdio.h>
#include <workbench/startup.h>

void main(int argc, union {
    char **args;
    struct WBStartup *msg;
} argv)
{
    int i;

    if (argc != 0)
    {
        printf("command = %s\n", argv.args[0]);
        for (i = 0; argc > 0; i++, argc--)
        {
            printf("argument %d = %s\n",
                i, argv.args[i]);
        }
    }
    else
    {
        printf("command = %s\n",
            argv.msg->sm_ArgList[0].wa_Name);
        for (i = 1; i < argv.msg->sm_NumArgs; i++)
        {
            printf("argument %d = %s\n",
                i, argv.msg->sm_ArgList[i].wa_Name);
        }
    }
}

```

rewind Reset a level 2 file position to its first byte

Synopsis `#include <stdio.h>`

```
void rewind(fp);
```

```
FILE *fp;    /* file pointer */
```

Description The **rewind** function resets the specified file to its first byte and is equivalent to the following **fseek** call:

```
fseek(fp, 0L, 0);
```

The **rewind** function also clears the error indicator on the file.

The **rewind** function is implemented as a macro that calls the **fseek** function.

Portability ANSI

See Also **errno**, **fopen**, **fseek**, **lseek**, **_OSERR**, **tell**

rewinddir Reset to the start of the directory

Synopsis `#include <dir.h>`

```
rewinddir(dfd);
```

```
DIR *dfd; /* dir file descriptor */
```

Description This macro changes the position from which the **readdir** function will read the next directory entry, resetting the current position to the start of the directory. The **dfd** argument is a directory file descriptor returned from a successful call to the **opendir** function.

The **rewinddir** function is guaranteed only for the life of the directory file descriptor. If a directory is closed and reopened, the directory entry locations may be invalid.

The **rewinddir** function is implemented as a macro that calls the **seekdir** function as follows:

```
seekdir(dfd, (long)0);
```

This function is not available if the **__STRICT_ANSI** flag has been defined.

Portability UNIX

See Also **closedir**, **opendir**, **readdir**, **seekdir**, **telldir**

sprintf Formatted print to a string

Synopsis `#include <stdio.h>`

```
length = sprintf(s,fmt,arg1,arg2,...);

int length;           /* number of characters generated */
char *s;              /* pointer to a character string */
const char *fmt;      /* format string */
typeargx              /* arguments */
```

Description This function produces an output stream of ASCII characters and sends the output to the storage area whose address is given by the argument **s**. Make sure that this area is large enough to hold the maximum number of characters that might be generated. The **sprintf** function also generates a **NULL** byte to terminate the stored string.

This function works like the **fprintf** function. See the description of the **fprintf** function earlier in this chapter for a complete description.

Portability ANSI

Returns This function returns the number of output characters generated. This number does not include the terminating **NULL** byte.

Example

```
/*
 * This example prints a message indicating whether
 * the function argument is positive or negative.
 * In the second printf, the width and precision
 * are 15 and 8, respectively.
 */
#include <stdio.h>

void pneg(double value)
{
    char *sign, buff[256];

    if (value < 0) sign = "negative";
    else sign = "not negative";

    sprintf(buff,"The number %E is %s.\n",
            value,sign);
    printf(buff);
```

sprintf Formatted print to a string
(continued)

```
        sprintf(buff, "The number %*. *E is %s.\n",
                15, 8, value, sign);
        printf(buff);
    }

    void main(void)
    {
        pneg(37.8);
        pneg(-18.2);
    }
```

See Also `fprintf`, `printf`

stacksize Get the current stack size

Synopsis `#include <dos.h>`

```
size = stacksize(void);

size_t size;          /* size of current stack  */
```

Description This function calculates the size of the current stack and returns its size in bytes.

Portability SAS/C

Returns This function returns the number of bytes available for the entire stack when the program was started.

Example

```
/*
 * Ensure at least 8K is available for certain
 * operations
 */
#include <stdio.h>
#include <dos.h>

void main(void)
{
    char *amount;

    if (stacksize() > 8000)
    {
        /* Do the operation */
        amount = "More than 8000 bytes";
    }
    else
    {
        /* Tell users it is not safe to do it */
        amount = "8000 bytes or less";
    }
    printf("%s stack available.\n", amount);
}
```

See Also `__stack`

stackused Get the amount of stack in use

Synopsis `#include <dos.h>`

```
size = stackused(void);
```

```
size_t size;           /* amount of current stack in use */
```

Description This function calculates the amount of the current stack currently in use and returns the amount in bytes.

Portability SAS/C

Returns This function returns the number of bytes of stack currently being used.

See Also `__stack`

strnicmp Compare strings, case insensitive, length limited

Synopsis `#include <string.h>`

```
x = strnicmp(a,b,n);

int x;                /* comparison result */
const char *a,*b;     /* strings being compared */
size_t int n;
```

Description This function compares two **NULL**-terminated strings using the ASCII collating sequence, but it does not distinguish between uppercase and lowercase. No more than **n** characters are compared.

The relative collating sequence of the strings is indicated by the sign of the return value, as follows:

Return	Meaning
negative	first string is below the second
zero	strings are equal
positive	first string is above the second

This function is not available if the **__STRICT_ANSI** flag has been defined.

Portability SAS/C

Returns The sign of the return value indicates the relative collating sequence of the strings, as noted above.

Example

```
#include <stdio.h>
#include <string.h>

void result(char *name, int r)
{
    char *p;

    if (r == 0)
    {
        p = "is equal to";
```


strnicmp Compare strings, case insensitive, length limited
(continued)

```

    }
    if (r < 0)
    {
        p = "is less than";
    }
    if (r > 0)
    {
        p = "is greater than";
    }
    printf("%s string A %s string B\n",name,p);
}

void main(void)
{
    char a[256], b[256];
    int n;

    while(1)
    {
        printf("Enter string A: ");
        if (fgets(a,sizeof(a),stdin) == NULL)
        {
            break;
        }
        printf("Enter string B: ");
        if (fgets(b,sizeof(b),stdin) == NULL)
        {
            break;
        }
        printf("Enter maximum compare length: ");
        scanf("%d",&n);
        result("strnicmp:",strnicmp(a,b,n));
    }
    printf("\n");
}

```

See Also `strcmp`, `strcmpi`, `stricmp`, `strncmp`

__tinymain Special version of the `__main` function

Synopsis `#include <stdlib.h>`

```
void __stdargs __tinymain(line);
```

```
char *line; /* ptr to command line that caused execution */
```

Description The `__tinymain` function is a special case of the standard C preprocessing function `__main`. It does not open the standard I/O files `stdin`, `stdout`, and `stderr`, and therefore, eliminates support for standard I/O routines from your executable code. If you use the `__tinymain` function, you cannot use standard I/O functions such as `printf` and `scanf`. Also, using the `__tinymain` function suppresses the initial input and output window normally displayed when starting up from the Workbench.

If you are not using registerized parameters, you can use the `__tinymain` function by specifying the following on the `slink` command line:

```
DEFINE __main=__tinymain
```

If you use the `stdio` option in the `sc` command, the compiler handles this for you. When you compile your program, the compiler adds an underscore to symbol names.

Portability SAS/C

See Also `main`

tmpfile Open a temporary file stream

Synopsis `#include <stdio.h>`

```
fp = tmpfile(void);
```

```
FILE *fp;          /* pointer to temporary file stream */
```

Description This function opens a temporary file. The name of the temporary file is constructed from a combination of the first three characters of the program name, a string of characters based on the base stack pointer, and a sequential number (*nn*):

```
T:progname.stack.T.nn
```

The **tmpfile** function attempts to open files of increasing sequential numbers until it succeeds, or it has tried 99 times. If you do not have the argument **T**: assigned, the **tmpfile** function will never be able to create a temporary file.

Portability ANSI

Returns This function returns a file handle for a file that can be read or written. When this file is closed, it is automatically deleted.

Example

```
/*
 * Create a temporary file to hold a sort data set
 */
#include <stdio.h>

void sortfile(FILE *fpo, FILE *fpi)
{
    /* Insert code for "World's Greatest File Sorter" */
    /* in this routine...                               */
    return;
}

void main(void)
{
    FILE *fp, *infp, *outfp;
```

XCEXIT Terminate the program

Synopsis `#include <stdlib.h>`

`void XCEXIT(lcode);`

`long lcode;`

Description This function terminates execution of the current program and returns control to the parent program. It does not close any files, but does call the standard termination routines.

 The parameter **code** is a value from 0 to 255 that gets passed back to the parent. By convention, a value of 0 indicates success. Normally, your program should call the `_exit` function unless it knows that no level 1 or 2 files are open.

Portability SAS/C

See Also `_exit`



**This was brought to you
from the archives of**

<http://retro-commodore.eu>